Author:
Foyer, Clement M

Title:
Abstractions for Portable Data Management in Heterogeneous Memory Systems

Abstractions for Portable Data Management in Heterogeneous Memory Systems

Clément Foyer

supervised by

Simon McIntosh-Smith
and Adrian Tate
and Tim Dykes

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of Doctor of Philosophy in the Faculty of Engineering, School of Computer Science.

June 2021

wordcount: 33 100

Abstract

This thesis is a study of data selection and placement in heterogeneous memories in modern high-performance computer architectures. Memory systems are becoming increasingly complex and diverse, which complicates the search for optimal data placement and reduces the portability of applications. As we enter the dawn of the exascale era, memory models have to be rethought to consider the new trade-offs between latency, bandwidth, capacity, persistence and accessibility, and their impact on performance. Moreover, this data management needs to be simplified and brought within reach of domain scientists in fields outside of Computer Science.

To address this issue, this work focuses on studying data movement, data optimisation and memory management in systems with heterogeneous memory. Firstly, a new algorithm was developed that improves the computation of data exchange in the context of multigrid data redistribution. Secondly, multiple APIs for memory management were unified into a single abstraction that provides memory allocations and transfers in the form of a portable, adaptive and modular library. Lastly, the allocation management was studied in a high-level language along with ways to enable low-level control over memory placement for a high-level language.

The Adjacent Shifting of PEriodic Node data (ASPEN) algorithm, presented in this thesis, provides better performance than state-of-the-art algorithms used for producer-consumer data redistribution of block-cyclic organised data, as used in distributed numerical applications and libraries (e.g. ScaLAPACK). The Mamba library was developed and aims to facilitate data management on heterogeneous memory systems. It uses a data broker developed with library cooperation and interoperability in mind. In addition to providing portability and memory abstraction, it also serves as a comparison tool for benchmarking or exploratory experiments. Finally, a use case of memory management in C for a Python application based on a distributed framework has been studied as a proof-of-concept for providing direct memory management to high-level application development.

This work presents a data-centric approach to the challenges heterogeneous memory creates for performance-seeking applications.

Dedication

First and foremost, I would like to thank Adrian Tate and Prof. Simon McIntosh-Smith for paving the way, guiding and accompanying me throughout the course of this PhD. I knew I could trust your lead and rely on your support; that fueled my motivation in the most difficult periods. I would also especially thank Tim Dykes for his continuous help, despite the jokes that made his eyes roll and my numerous questions and interruptions. This work would not have gone so well without your assistance.

I also thank the HPE HPC/AI EMEA Research Lab team for welcoming me in the team, for their advice, humour and kindness. It made the everyday work much more enjoyable by interacting with you. There isn't a day that I would have rather stayed at home and I am looking forward to collaborating with you again. Many thanks as well to the HPC Research Group at the University of Bristol. Thank you for making me included, despite the little time spent there, and explaining the arcanas of university when I needed it, because of the little time spent there.

A global thank-you goes to the friends I met through the EXPERTISE project — especially Pierlauro and Hatem —, to the too-numerous-to-be-named friends I made in Bristol, and to the former colleagues and still friends I interacted with on IRC. You kept my spirit high, my social life balanced and my mind sane. So long, and thanks for all the fish.

I address a special thank-you to Kathryn and Jim for the many proof-readings, despite not always understanding the meaning because of the technicity of the topic or because of the disastrous syntax. You helped me greatly.

Finally, I cannot forget to thank Cédric, for always being there to support me, to pick me up or drop me at the airport, and sheltering me whenever it was needed. I shan't forget to thank my family, especially my parents, for their love and support during my education and for giving me my values, shaping the way I see and interact with the world; you made me who I am, and my resilience is the result of your support and education.

It is the help and influence of each and everyone of you that not only made this thesis possible, but made it a joyful journey.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: DATE:

Acknowledgements

This work was funded by the EXPERTISE project[1] and EPiGRAM-HS projects[2], which have received funding from the European Union's Horizon 2020 research and innovation program under the Marie Skłodowska-Curie Grant Agreement № 721865 and Grant Agreement № 801039 respectively. This work also benefited from access to MareNostrum and Nord3 provided by Barcelona Supercomputing Center, thanks to Rosa M. Badia.

[1] http://www.msca-expertise.eu/
[2] https://epigram-hs.eu/

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

Chapter 1

Introduction

Contents

1.1 Contextualisation

In June 2020 the Supercomputer Fugaku built by Fujitsu delivered for the first time more than one ExaFLOP/s in single precision performance and signalled to the world that we are entering the so-called exascale era. What is significant about this particular phase of computation is that the physics and thermal properties of our computation devices are constrained in ways that create new challenges and roadblocks to progress. Primary among those challenges is the management of memory and data. To deliver the peak performance, data needs to be quickly accessed, or streamed in order to be efficiently browsed through. The volume of data is also increasing with the convergence between HPC (High-Performance Computing) and Big data that makes HPDA (High-Performance Data Analysis). Hence, the frontier between memory and storage is fading with memories reaching terabytes of capacity while storage is getting disaggregated and staged across nodes or burst buffer.

Furthermore, with an increasingly complex memory system with heterogeneous characteristics (latency, bandwidth, persistency, etc.), the data management requires more attention in order to reach the best performance out of evermore expensive supercomputers. The processing systems are diversified as in addition to the CPUs and GPUs emerge FPGAs and specialised processing units like Graphcore's IPUs that are dedicated to AI. Memory technologies are similarly evolving as high-bandwidth memory is becoming standard in GPUs while persistent memory with standard DRAM performance emerges on compute nodes.

In order to achieve the highest performance, the data location in memory needs to be carefully chosen, depending on access frequency, data dimensions and the arithmetic intensity per byte, for considering to optimise the latency, the capacity or the bandwidth, respectively, for example. Moreover, those parameters may not be static during the application's execution. This implies that some data need to be replaced or moved to provide space for more critical data. Thus it is of utmost importance to provide ways to efficiently select data as a whole or subsets of it, and to deal with specific

APIs that accompany specific memory tiers. Finally, many libraries only provide a support for a limited number of programming languages, which may not overlap with the set of languages used by the majority of the scientific community. Additionally, the managing of memory requires knowledge about the machine (e.g. hardware used, number of packages, processors, etc.) and the way it works in order to understand how to use it to its best (e.g. impact of cross-NUMA-domains memory access).

This thesis addresses these questions in several different ways. It presents the study of redistribution algorithms between two regular N-dimensional grids of data. Given a block-cyclic distribution P, the objective is to compute the size and offset of chunks of data to exchange in order to transform the distribution to a distinct block-cyclic distribution C. This kind of algorithm is often used in the context of distributed memory, but given the increasingly disparate nature of memory systems, the study of such an algorithm was deemed pertinent in the context of memory tiling, for example.

A second aspect of data management studied is more practical. The review of memory systems interfaces led to the observation that using heterogeneous memory systems requires either a limitation on the language to use for programming, or a limitation on the set of compilers available, and often requires the adaptation of every call to the specific libraries' API. So it requires a high level of expertise which could be alleviated by providing a unique interface to memory. Moreover, the development of such an interface associated with a memory abstraction layer would provide a reflective tool to explore the memory state during the execution of an application.

A final aspect of data management was the accessibility across programming languages. Not all languages provide interface for memory management, but many provide some way to interface with a programming language that provides such support. Languages such as Matlab or Python could benefit from a manual memory management that would interact with broadly used libraries such as NumPy. The work completed on the PyCOMPSs framework to provide shared-memory capabilities in order to reduce the number of I/O operations was an interesting case of study in providing low-level memory management (shared-memory) to high level

language (Python).

In summary, this thesis addresses three independent aspects of the memory management in modern HPC practices and studies the practicality of portable solutions based on system abstraction.

1.2 Contributions

The work performed in the course of this thesis led to the following contributions:

- Memory systems technologies currently used in the HPC industry have been surveyed. Along with the different approaches to memory management, this provided a valuable insight into the state-of-the-art that was used for determining the more suitable approach for our objective of abstraction and portability.

- The ASPEN algorithm has been developed. It provides a new way to compute complete redistribution of data across distributed agents. It uses periodicity and considers data organisation to provide improved performance compared to industry standard like ScaLAPACK.

- The ASPEN approach has been included in UDJ (Universal Data Junction), a library developed in the HPE HPC/AI EMEA Research Lab for portable distributed data exchanges. The algorithm is used to compute the offset and size of data to exchange in order to minimise the size of data transfers.

- A new memory abstraction has been designed, accounting for the heterogeneity of memory systems, but also the heterogeneity of programming environments. The plurality of the frameworks available is necessarily taken in account to provide a solution that fits the user requirements. The design focused on portability and abstraction. The resulting library, Mamba, enables a data management of tiled arrays, with a tracking of each independent tile.

- In addition to the design of the memory abstraction, the appropriate mechanisms has been developed as part of the MAMBA library to support the tile management. Particular care was given to the integration of already existing libraries to benefit from their specificity, but also to broaden the scope of utility of the proposed data broker. Performance was evaluated with four different micro-benchmarks based on a standard library performance evaluation.

- An additional feature has been included into PyCOMPSs. The distributed task-based Python framework has been enriched with support for intra-node data-sharing with minimal code alteration. This feature improves the performance of tasks by increasing the data reuse which avoids superfluous disk accesses. The performance gain has been assessed with two types of application representing different kinds of data-access patterns, namely k-means and matrix multiplication.

These contributions constitute the result of a multifaceted approach to the question of coordinating large amounts of data on a multiplicity of newly released hardware, for a selection of programming models and programming languages.

1.3 Thesis outline

The remainder of this manuscript is split into four chapters, each focusing on one aspect of the research conducted, with an extra chapter to conclude this work.

Chapter 2 summarises the state-of-the-art regarding memory systems. In this chapter, the basics of memory are presented and explained as well as the impact the different systems have on computational performance. The different technologies, present and past, are introduced with an emphasis on the difference and advantages of each. Finally, different

families of memory management will be introduced (programming models, frameworks, libraries), along with some of the most remarkable examples.

Chapter 3 presents the earliest work done on the topic of memory management. The question of selecting data for effective data movement in the context of the redistribution of regular grids. In this chapter, a new algorithm for regular grid redistribution named ASPEN is described. This algorithm is compared with state-of-the-art algorithms showing better performance.

Chapter 4 focuses on memory management for applications in the context of heterogeneous systems. This chapter presents Mamba, an array-based abstraction for heterogeneous memory. First, the high-level library is succinctly outlined before providing details about the structure of the modular library for memory management. In addition to providing support for heterogeneous memory systems, this library may be used as support for benchmarking and parameter exploration. The performance and modularity of the tool are presented and evaluated in that chapter as well.

Chapter 5 introduces the work done as part of the secondment at the Barcelona Supercomputing Center. As part of the collaboration between the member of the EXPERTISE European project consortium, the objective was to study the feasibility and potential gain of providing memory sharing capabilities for a complex Python framework. To that end, different options were studied and a module was developed and integrated. This chapter presents the results, showing the possibility of seamlessly integrating low-level fine grained memory management into a high-level language such as Python.

Chapter 6 wraps up the work presented in the document, outlining the different questions that were addressed. This chapter will be dedicated to linking the different questions before describing the new research opportunities that have been enabled by the work presented thereafter.

Chapter 2

Memory Landscape

Contents

Computing systems are also called information systems and information has to be stored first before being processed. Information refers both to the data processed and to the actual description of the processing (i.e. the program in its binary form).

This chapter presents all the prevalent notions which relate to memory systems in the context of computer science, especially in the narrow field of HPC, both from a hardware and from a software perspective. Section 2.1 introduces the questions that have been tackled and the answers that were provided; Section 2.2 will be dedicated to presenting the current approaches that aim to facilitate the usage of heterogeneous memory systems; Section 2.3 will summarise an analysis of the state-of-the-art and present a proposal for a solution for the completion of the missing pieces.

2.1 Memory systems review

The Turing Machine defines an abstract machine, which manipulates symbols on an infinite strip of tape, divided into discrete units, according to a table of rules. Information systems are a complex evolution of Turing Machines, but the presumption of *infinite* memory persists in the modern era. Since the 50's it has been clear that information systems would need more memory than the small batch available in the core; hence the usage of replaceable drums of memory that enable a virtually infinite renewal of bytes. However, when the size of the problems solved increased, it became clear that the handling of physical devices would become problematic. Also, a manual management of data locality would be too impractical for substantial programs. In addition, it would be impossible to write programs without assumptions about the size of the memory used.

In order to be able to abstract the hardware, the data had to be automatically managed and moved across levels, transparently to the programmer. This section will present the different structures that have been added one to another across time to answer the question *'How to provide infinite memory to computing systems without penalising either correctness or efficiency?'*

2.1.1 Evolution of processing speed versus memory speed

For the last forty years, the growing discrepancy between computational and memory performance has been a well known issue. Addressing it has been delayed until the slowdown in computing performance could not compensate for the lateness in memory management anymore, despite the best efforts in cache management and organisation. As presented in Figure 2.1, we can see two separate levels. While the amount of computation to be executed steadily rose, the rate at which data were available to be processed rose at a much slower pace. The improvement in terms of computation depends on several factors. The clock frequency that orchestrated the operations in the processor rose to 5 GHz before stabilising around 3–4 GHz, and is not expected to exceed the latter [41]. But in addition to being able to execute more operations per second, since the early 2000's the number of cores increased, enabling independent streaks of instructions to be executed at the same time. Also, the complexity of operations increased. For example, the appearance of vector operations enabled the execution of the same complex computation up to 8 elements simultaneously. Finally, the instruction execution process has been decomposed, staged and potentially reordered to reduce the stalling between operations even more. [78, 105]

However, in order to feed such an appetite with data to crunch, an ever increasing data pipeline is required. Trying to reduce the increasing gap between computing performance and memory latency, as shown in Figure 2.1, the architecture of processors has been made increasingly more complex as a solution to solve the *memory wall problem* [138].

The data buses have been widened, but the improvement in the number of pins since the 80's has not allowed the bandwidth to grow sufficiently. In order to compensate for this, and to reduce the impact of memory being more remote and slower to reach, extra space needs to be found in silicon in order to provide a quick access to memory. But the package cannot contain the hundreds of billions of bytes that may be required for applications. In addition, nor can the whole program, in its stream of instruction form, be

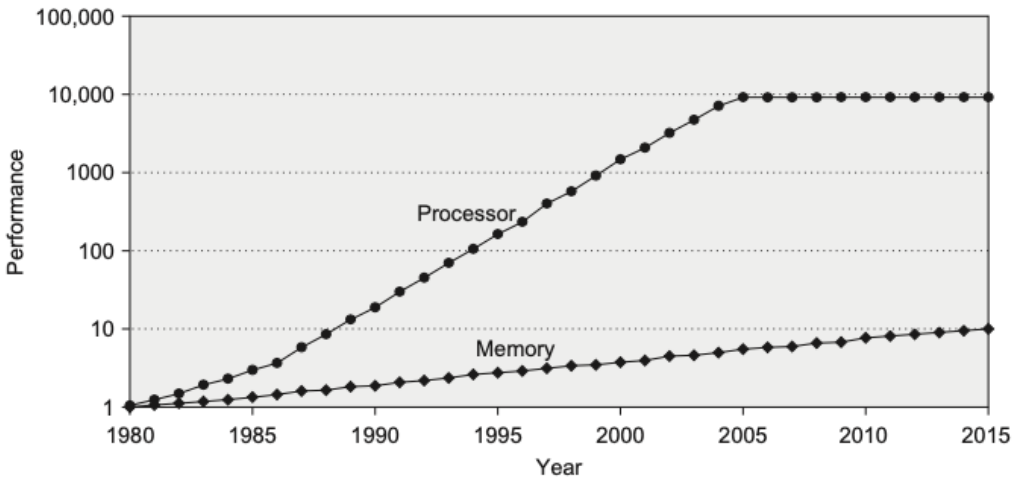
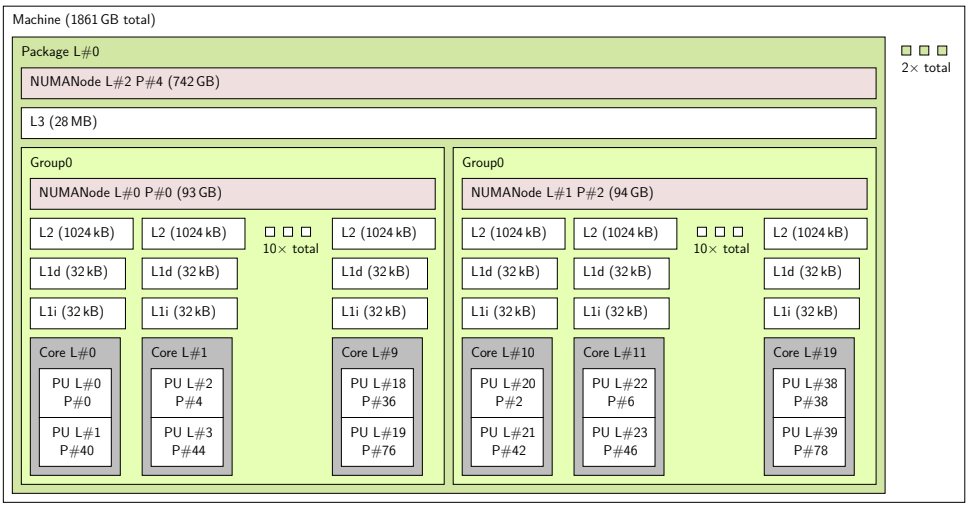


Figure 2.1: Performance gap between computing power and memory latency (from [110]).

contained in the small memory embedded into the package. Hence, the access and interaction with the memory systems are inherently critical for many applications and the granularity of data access has to be adapted to the resources available. This also implies that the data loaded into the package memory has to be renewed regularly, adding complex questions about the frequency of updates, which data to load next, and which data to remove [133]. To answer the question of which data to load, the following two assertions are generally made:

1. *The latest datum accessed is likely to be accessed again*;
2. *The next datum to be used is located nearby the last datum accessed.*

As computers have been developed in order to resolve many iterative and repetitive tasks, these lemmas are mostly true. Moreover, programmers aware of the issue and eager to get the best performance out of their systems are likely to consider this matter when creating their algorithms. Hence, they will be trying to reuse data as much as possible and whenever possible, doing sequential operations making the previous two strategies a self-fulfilling prophecy. These two lemmas are known as the *Temporal locality*



The displayed machine exposes two Intel Xeon 6230, *Cascade Lake* architecture, each with DRAM (NUMA nodes $L\#0$ and $L\#1$) as cache of NVDIMM (NUMA nodes $L\#2$). L3 cache level is shared between the two groups of processors, while L1d, L1i and L2 caches are exclusive to each core.

Figure 2.2: Example of complex architecture as displayed by **`lstopo`** [58].

principle and the *Spatial locality principle*. As a matter of fact, in addition to staging the memory across different subgroups, each subgroup also has its specific rules and characteristics. For example, Figure 2.2 shows the extremely hierarchical architecture of Intel's latest processor and its integration with the heterogeneous memory system. This shows one of the evolutions made in order to improve the performance of processors. For example, one processor may have multiple hyper-threaded cores, where two execution pipelines share the resources in one core.

However in order to provide enough data throughput to feed instructions, some smart architecture design decisions have been made. First, the different levels of caches correspond to different *distances* between the memory group and the processing unit. The bigger the number, the bigger the memory capacity, but the further from the processing unit the memory will be located. In addition, each cache level has a specific set of cache policies depending on how many processing units access it, how redundant or how spread out the data are. Examples of such policies will be presented in Section 2.2.1. Caches can also be dedicated to contain only data or instruc-

tions as each follows a specific control flow with regard to the execution of a program. The caches get their data from the main memory or from devices whose architecture will be outlined in Section 2.1.2.

There are different metrics to be taken into account when evaluating the characteristics of memory. These characteristics will be spelt out in Section 2.1.3.

2.1.2 Current technologies

The technology used today for main memory is typically DRAM. Although many flavours of DRAM are available, usually the SDRAM (Synchronous Dynamic Random-Access Memory) one is preferred for main memory. In this section will be presented the internal structure of a memory module that retains data and the implications it has on memory access performance.

2.1.2.1 DRAM

DRAM is the most common format for big capacity in a compact form factor, for a good price with a quick reactivity. It is used widely on package (e.g. eDRAM on [79]), close out-of-package memory (e.g. HBM2 in A64FX processors [122]) or as main memory in dual in-line memory modules (DIMM). The internal organisation is the same in most cases, but the names used in this section correspond to the latter case.

2.1.2.1.1 Memory cell As shown in Figure 2.4-b, one memory cell is usually composed of one transistor, to manage accesses, and one capacitor, to store the bit[1]. The charge in the capacitor determines whether the memory cell is set to 0 or 1. However, because of the electronic nature of this component, some leaking occurs over time, which means the memory cell has to be refreshed. A second drawback is that reading a value is destructive, so the memory cell has to be rewritten after being read, operation

[1]Most frequent architecture nowadays, although the original designs had three or four transistors; capacitor-less memory cells have been developed, but are not widely used.

done by the sense amplifiers. Hence, there is a necessary cool-down period after reading a bit before being able to read it again as well as a periodic unavailability due to the memory refresh period.

2.1.2.1.2 Bank The bit-cells are organised and interconnected in a 2D grid called a *memory array*. More precisely, the bit cells are grouped in cache lines, which are then laid out in rows and columns. This organisation allows for quicker accesses, as each bit is not addressed independently, and simplifies the circuit for sending groups of bits at once back to the memory bus. The memory refresh circuit of each bit of the same column can also be shared, updating a whole row at once, one row at the time. All these circuits are packaged together in a bank. In today's systems, banks are the smallest memory structures that can be accessed in parallel with respect to each other [77]. Each bank is smaller than the entire memory storage and access to different banks is independent. This independence can improve the latency with the right interleaving of data across different banks.

2.1.2.1.3 DRAM chip Banks are grouped together in a DRAM chip, commonly by eight units. Within a chip, all the banks share the command, address and data bus in order to simplify the management of the independent, yet synchronised, units. To access data, the chip activates one bank that reads the requested row from the row-address bus. The row is charged into the row buffer (also known as sense amplifier) with the ACTIVATE command. The column-address is read to select and to read all or part of the row that will be sent to the DRAM data bus. DRAM chips usually have a narrow interface, 4 to 16 bits per read. [110]

2.1.2.1.4 Rank In order to provide a wider *memory word* while keeping the parallelism and independence in data management, the DRAM chips are assembled into a rank. Although working synchronously and responding all to the same address, each DRAM chip provides different data. It is the concatenation of the data coming from each of the DRAM chips that creates the memory word that is being returned to the processor by the memory

module. For example, for a 64-bit memory word, with DRAM chips having a 8-bit interface, one rank would have 8 chips, each providing one byte of data, therefore allowing 8 bytes to be read in a single access.

Ranks also enable a simplification of the memory controller as there is no need to deal with each individual chip. However, it comes at the cost of a higher granularity: there cannot be any access smaller than the interface width.

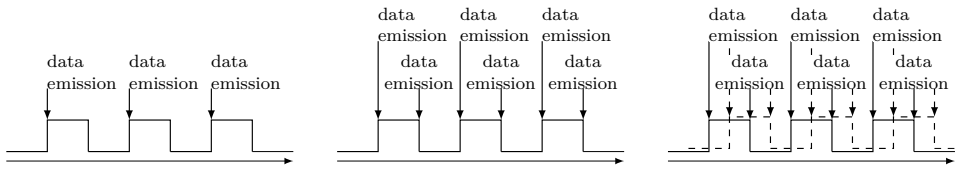
Finally, modern DRAM modules are composed of one or multiple ranks. For example, DIMMs are most commonly used nowadays, replacing SIMMs in the early years 2000. [110] Dual in-line memory modules are connected to the PCIe bus, receive the command and transmit it to the corresponding rank. All the ranks on a single DRAM module share the address and the data bus, only one being allowed to use them at one time.

2.1.2.1.5 Channel The is the final granularity when interacting with SDRAM. The channels are subdivisions of the memory bus. They can be either independent, in which case each requires a dedicated memory controller, or dependent (or *locksteps*) with only one controller but with a wider interface.

2.1.2.2 Data rate

For synchronous memory modules, the data rate defines the way the signal of the external clock triggers data movement, as illustrated in Figure 2.3.

For SDR (Single Data Rate), only the rising edge of the clock signal is considered by the memory module. There is only one word of data transferred during each cycle. For DDR (Dual Data Rate), both the rising and falling edges trigger data emission, doubling the amount of data transferred for each cycle. There is also a QDR (Quad Data Rate) that has been used in the 90's and 2000's for Intel's Front-Side Bus and later by IB (InfiniBand), as an example, but it has not been widely used for main memory. [110] In QDR, a second clock signal is emitted, 90° out-of-phase from the first signal. The memory module emits bytes in synchronisation with the rising and falling edges of both signals.



(a) SDR (b) DDR (c) QDR

Figure 2.3: Examples of bytes emission for each data rate.

For completion, this section also presents GDDR (Graphics Dual Data Rate) which is a specialisation of DDR with which it shares the same core technologies. However, the GPU dedicated memory works at a higher frequency with a different protocol that allows for better bandwidth, lower power consumption and lower heat dissipation than DDR.

The specifications of these interfaces are defined by the JEDEC. The latest version specified widely used are DDR4, defined in [4], and GDDR6, defined in [3]. DDR5 has been released in July 2020 [14] by the JEDEC. It was designed to meet increasing needs for efficient performance in a wide range of applications including client systems and high-performance servers [15]. DDR5 supports double the bandwidth and is expected to be launched with a data rate 50 % higher than DDR4's end of life speed. The power efficiency is also improved with reduction of 10 % of the required voltage for the field effect transistors. Finally, NVDIMM-P was introduced in November 2020 [69] and is expected to provide improved performance. This new channel protocol is trying to enable the combination of persistence and large capacity together, with limited to no cost on performance. It is expected to be used in channels in combination with deterministic accesses such as DRAM [13, 57].

There are many other interfaces that are optimised for different purposes (low-power, high-bandwidth, low-latency, 3D-stacked, etc.), yet the underlying microarchitecture is fundamentally the same. If a flexible memory controller can support various DRAM types, it singularly complicates it and makes the micro architecture hard to maintain and upgrade.

2.1.2.3 NUMA node and NUMA-effect

The operating system basic abstraction when it comes to identifying memory units is called a NUMA domain. The acronym NUMA stands for Non-Uniform Memory Access as it expresses the variability of memory characteristics, such as latency or bandwidth, depending on where the code is executed. This variability is called the NUMA effect. One NUMA domain is exposed as a NUMA node corresponds to a specific set of performances that are common for the considered memory unit considering one computing unit. For DRAM, it usually corresponds to the multiple channels connected to the same memory unit. Because this memory unit is often included in the processor package, for performance reasons, this abstraction also corresponds to the processor socket, for simple cases. When HBM is available, because its characteristics are different from the DRAM's, it is exposed in its own NUMA node even if it shares the same memory management unit. For example, Intel's Xeon Phi *Knight Landing* topology exposes two NUMA nodes per group (in *flat* or *hybrid* modes), one for DRAM and one for MCDRAM [124].

2.1.3 Intrinsic memory characteristics

Depending on the technology used, memory expresses different characteristics, the two prominent ones from a software high-level abstraction point of view being the latency and the bandwidth. These two notions shall be further introduced in the following Section 2.1.3.1 and 2.1.3.2, respectively. The other subsections present a brief introduction to the internal factors impacting a memory module performance.

In recent years, as part of the run toward exascale, other characteristics have emerged for DRAM. Persistency (as opposed to volatility) has become increasingly interesting for application developers as it offers a new device for temporary data storage, either for check-pointing or communication buffering. It also often comes with a substantially higher storage capacity than standard DRAM.

These attributes show much variability depending on which type of

memory is referred to (e.g. SRAM or DRAM; SIMM or DIMM; SDRAM, HBM or NVDIMM; etc.).

2.1.3.1 Latency

The latency for an information system is the time between the request of an access to some data and its availability for use. For applications that often need to access data in many different parts of memory, like a key-value storage system, optimising this characteristic is critical. As previously exposed, the general rule of thumb for data availability is to access data that are located closely to each other in order to facilitate the work of the *data prefetcher*.

When requesting data one of five cases can arise. Either the data are already in the processor and the operation can proceed. The second case is having the data available in one of the caches, in which case a few nanoseconds later, or ten or so at most, the data are available and the operation can proceed. The third case would be having to reach all the way to DRAM to find the data, in which case the data would have to be transferred to the caches, taking hundreds of nanoseconds, then to the processor core before the operation can proceed. The fourth case, would be to have the data out-of-reach, swapped to the disk, in which case it would take at least milliseconds to load back to DRAM and reach the previous case, where the operation can proceed. Finally, the last case would be to realise that the data are not available anywhere and does not exist, in which case the operation cannot proceed.

From this description we can determine that location is an impacting factor for memory latency. But a more detailed approach can also show finer disparity between memory systems. For example, the organisation of the RAM in DIMMs creates multiple latency factors. When the request reaches the DIMM, if the row is *open*, selected within a bank, only the reading of the column, Column Address Strobe (CAS), needs to be added to the bus latency when the transfer starts. Otherwise, first the correct row has to be loaded (Row Address Strobe (RAS)), then the CAS latency

is added to the bus latency. But if the wrong array was charged then an additional latency is added for the PRECHARGE which resets the bank to a state where a row can be charged. Overall the worst case scenario latency for DRAM can be expressed as PRECHARGE + $RAS + CAS +$ *bus latency*.

Finally, because DRAM technology uses a capacitor, there is a natural *leakage* of electric charge that obliges each row to be periodically refreshed. This process requires all the rows of the bank to be charged, read and rewritten at regular intervals. During this time, the impacted memory cannot be accessed for reading or writing, in addition to presenting the disadvantage of being fairly costly in energy. There are two ways the DRAM can refresh. Either the whole bank is refreshed at once, in a *burst* refresh, which will refresh all rows, one after another; or the refresh is *distributed* and each row will be refreshed at a different time, at regular intervals. The first case makes the whole bank unavailable for some time but is entirely available the rest of the time, while the second method makes the bank partially unavailable more often, but for a much shorter time frame. The issue is well known and has been studied in order to evaluate, measure and mitigate the cost of DRAM refresh on performances [27, 87, 88].

2.1.3.2 Bandwidth

Bandwidth defines the data throughput that can be achieved when reading or writing bytes to or from a memory (e.g. DIMMs, disk, network, GPUs or caches). Its importance is very high when it comes to optimising the overall system performance. Although memory access patterns are influential on the ability of a program to execute of current highly parallel processor at its peak performance[2], multiple hardware factors can also influence the ability for a processor to renew its cached-data at sufficient rate to keep the core pipeline free of stalls. Multiple factors influence this parameter.

The first and most obvious one is the number of memory pins dedicated to data transfer. Available memory bandwidth is limited by the number of wires over which data can be transferred, which is, in turn, limited by the

[2]See Section 2.3.1.3 §*Memory access pattern classification*.

number of direct connections that can be made. However, the area available on the processor's interposer, which connects it to the outside world, limits how many of such connections can be made. Hence, the number of direct connections between the peripheral modules and the processor are bounded by the surface availability on the processor's interposer.

The second factor is the memory frequency and the data rate used to communicate with the DIMM. The first parameter corresponds to the clock rate of the memory module, which synchronises the access to the banks' rows and their bytes stored. The second is dependent on the technology used for the DRAM memory controller, and allows for much improved performances as, for example, DDR allows for twice as many instructions to be executed on the memory module, and up to four times as many for QDR relatively to SDR. However a higher frequency does not necessarily mean the effective throughput will scale. Because of the internal parameters such as the data interleaving across the different banks, some row-level conflicts may appear, sequentialising otherwise parallel memory accesses [77].

Additionally, DRAM banks may apply extra timing penalty if two rows are being accessed sequentially from the same bank as the bank must get back to the PRECHARGED state before being able to load a new row. In order to use the memory bandwidth to its maximum capacity, the data accessed must be stored on different banks, from different ranks, from different channels. The current standard for DDR4 is 25.6 GB/s peak target bandwidth, and DDR5 is announced to double the bandwidth, with a peak value of 51.2 GB/s. [4, 14]

2.1.3.3 Capacity

As expressed in the beginning of this chapter, the computing systems theoretically consider the memory to be infinite. However, there are multiple limitations to such an assumption. First and foremost, in a finite world with finite resources, there can be no such thing as infinite memory. More pragmatically, computers use a limited set of addresses, which can uniquely identify data. In most modern machines, the granularity of addressing is

the byte. In modern architectures, addresses are expressed on 64-bit words, allowing for 2^{64} uniquely identified addresses; it is hence possible to discriminate between 18 446 744 073 709 551 616 different bytes. Although the question of limiting the range of addresses not only comes from the range of values that can be represented with one 64-bit memory word, but also the number of unique addresses that can be requested from the DRAM, given the bus size. Thus, the memory system capacity for computers, disregarding the storage system, is usually limited by three factors that are intertwined.

The first one is the physical space. The processor basal substrate is limited in size. In addition to having to develop specialised hardware, the energy required for the system to work, and the heat dissipated would limit the potential gain in performance. Heat is the second factor. Finally, the bigger the memory, the longer it takes to determine the location of any specific data. Hence, modern architectures tend to specialise memory, disaggregating it, and developing efficient and ingenious systems to move the right string of bytes to the right location at the right moment, leaving the capacity issue to slow, external devices.

2.1.3.4 Volatility

There has been a growing interest in this characteristic in the past 10 years. The issue of having memories with big capacity but slow access may impact performances in multiple aspects. First, it means that data first have to be loaded from the *storage* memory to the *working* memory for computations to be executed. In addition, it makes systems very sensitive to failures and errors. To mitigate the risk during lengthy computations, the application may save some partial results[3], but this has an ever increasing cost as the application sizes get bigger along with the supercomputers. That is one example where providing non-volatile memory[4] would be particularly convenient and could lead to an improvement in performance. Multiple

[3] See Section 2.1.4.2 §*Checkpointing*.

[4] Memory systems where the data persist unaltered, even after the system has been turned off.

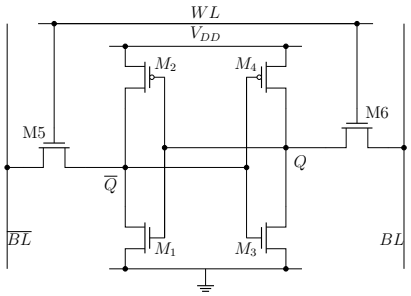
technologies have been developed and studied to provide such high capacity non-volatile memory with DRAM-like performances. They will be presented in the Section 2.1.4.2.

2.1.3.5 Endurance

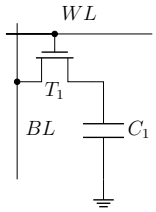
This characteristic expresses the durability of memory cells, i.e. how long they will work before they no longer are capable or retaining data. Although it has not been an issue for most historical technologies, since the emergence of Flash technologies, the question started to be raised. Flash memory, when written, has to erase and rewrite an entire block of data. In addition, the erasing and rewriting cannot be done on site but requires a memory block to be available on the device which increases the frequency of accesses of the memory cells. However, because of their architecture, the cells are much more sensitive to wear and tear. The industry norm for NVM is a typical retention lifetime of 10 years [18, 92, 94]. For Flash memories the endurance for 10 years corresponds to withstanding typical 10^6 cycles without suffering reading, writing or erasing failures [92, 94].

2.1.3.6 Density

This characteristic defines the number of memory cell that can be assembled to create a bank. Its unit is an arbitrary area unit which is independent from the technology used and the resolution used for the lithography. In volatile main memory, DRAM must not only place charge in a storage capacitor but must also mitigate sub-threshold charge leakage through the access device (see Figure 2.4). Capacitors must be sufficiently large to store charge for reliable sensing and transistors must be sufficiently large to exert effective control over the channel [82]. Scaling beyond 20 nm has been challenging, notably because of the diameter of capacitors, and technologies hardly managed to move forward by incremental steps of 1 or 2 nanometres.



(a) 6T-SRAM single port [9]



(b) 1T1C-DRAM [16]

The SRAM is composed of 6 transistors (4 for the storage, 2 for the access); the DRAM is composed of 1 transistor and 1 capacitor (transistor for the access and capacitor for the storage). WL is the *word-line* which select the appropriate row, and the *bit-line* (BL) retrieves the state of the bit stored.

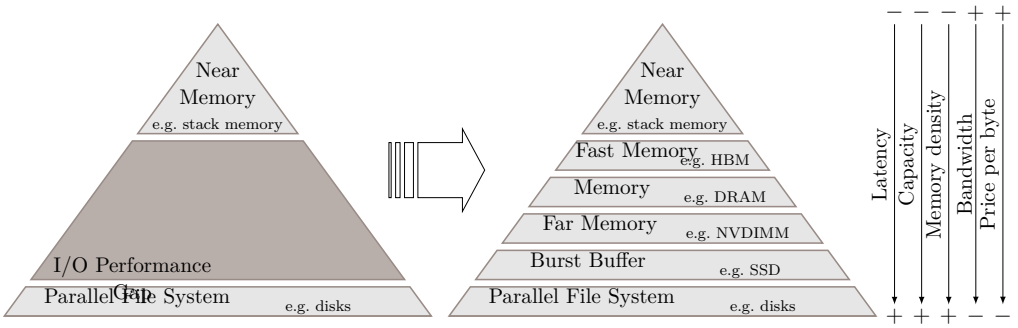
Figure 2.4: Examples of Static Random-Access Memory and Dynamic Random-Access Memory structures to compare the number of transistors required.

2.1.4 The emergence of new memory technologies

Although somewhat far from the studied problem, memory technologies can have an effect on the way we use them. Because of their intrinsic characteristics, the time before accessing the data is variable, and so is the expected gain for using them [111]. Any change in memory hierarchy affects programming models and code optimisation, so is therefore complex [41].

Many emerging technologies have been explored to provide new ways of mitigating memory access costs. Notably, the past years have seen a renewed interest in the technologies providing non-volatile capabilities. This interest is due to the need to close the gap between close memory and storage-class memory. The different technologies and their usage are usually still very hierarchical, and can be represented as in Figure 2.5.

Two approaches have been studied in parallel to close the performance gap between close memory and storage. It could roughly be divided between computation-oriented optimisation and storage-and-network-oriented optimisation. Both trends have their specificities that will be explored and



Inspired from [39, Fig. 1] and extended.

Figure 2.5: An example HPC storage hierarchy to fill the I/O performance gap between main memory and disk.

outlined in the Section 2.1.4.1 and Section 2.1.4.2.

2.1.4.1 High Bandwidth Memory

DRAM versatility is praised and it is a surprise to no-one that it has been so hard to try to find a replacement, or even a complementary technology to help reduce the gap between computation performance and memory access performance. One idea is to try to change the form-factor of the memory. Based on normal DRAM technologies, but with 3D stacking to improve the density, located close to the processor, sharing an interposer, to improve the bandwidth. Multiple technologies have been studied in the past 10 years in order to close the gap between LLC and the DRAM. Although they are all known as High Bandwidth Memory, this denomination corresponds to the JEDEC defined standard cited in [5].

It started with the study of HMC (Hybrid Memory Cube). Its memory organisation is made vertically, with the access-unit being the *vault*. Each vault corresponds to a DRAM channel in terms of independence of data transfer. The package is composed of 4 or 8 die plus a logic base, all stacked. In the example presented in [95], with 4 stacked die, it behaves as if we had 16 channels, each with 4 ranks. The links between the package and the memory banks are divided into 16 full-duplex lanes. Overall, this technology allows up to 8 GB of memory and a bandwidth of 160 GB/s.

HMC with 8 die doubles the overall memory, and would correspond to 16 channels with 8 ranks of DIMMs. The 3D structure uses through-silicon vias (TSV) [30] in order to communicate the bytes to the interposer to reach the caches, and ultimately the processing units, more efficiently. The advantages of die stacking are a lower voltage requirement, a simplified management of memory access, a better capacity than the caches and a 9× improvement factor of the bandwidth compared to DIMM[5] as the vaults act like as many DRAM channels.

A similar structure has been used, for example, for MCDRAM (Multi-Channel Dynamic Random-Access Memory), the HBM developed in collaboration with Micron Inc. for Intel Xeon Phi *Knights Landing* (KNL) chips. The chips were embedding 8 stacks of 2 GB of HBM, on package. The memory could either be managed by the operating system and the chip driver as a LLC, and be addressable directly by the user for a manual management of it. All eight MCDRAM devices collectively provide an aggregate bandwidth of more than 450 GB/s [124].

New features have been introduced in HBM2 [71], such as pseudo channels and implicit precharge operations, as well as storage with error correcting code (ECC). Pseudo channel mode is a very important improvement in HBM2. In the pseudo channel mode, each 128-bit channel can operate as two separate pseudo channels of 64 bits. A pseudo channel consists of 4 bank groups where each group has 4 banks. By using the pseudo channel mode, HBM2 can achieve optimised command bandwidth, decreased latency, and higher effective data bandwidth.

In Fujitsu A64FX processors, the HBM2 is directly connected to the cores and L2 caches via the Core Memory Group (CMG), but located outside the package. The processor is connected to 8 devices of 4 GB each, for a total of 32 GB and a cumulated bandwidth of 1024 GB/s [122]. The HBM2 standard authorises up to 24 GB of memory per stack, and a bandwidth of 307 GB/s.

Overall, the much improved performance enables more efficient data transfer between an intermediate memory and the processor, but the rela-

[5] HBM2 compared to DDR5, as reported in [10].

tively small capacity makes it impossible to make the DDR-DRAM disappear entirely.

2.1.4.2 Non-Volatile Memory

Non-volatile memory can be differentiated into two groups, depending on the technology used, as defined by JEDEC. NVDIMM-F offers non-volatility using Flash technology for the storage. It also provides directly addressable NAND-Flash which is accessed as a block-oriented mass storage device [11]. NVDIMM-N resembles it, except that the NAND-Flash memory is embedded in the memory module, along with a small backup power source [12]. In the event of a power down, the power source is used to copy the data from the DRAM to Flash. The data are copied back when the power is restored.

2.1.4.2.1 Current and future technologies Although solutions based on Flash technology are the first available, a few alternatives have also been investigated recently. The first reason was that the latency of Flash system is much higher than the one expected of standard DRAM. A suitable candidate for a proper NVDIMM must be able to provide equivalent performance, be byte addressable for both reading and writing, and expose a suitable life expectancy.

Other desired traits may be an increased byte density, allowing for an increased capacity, and a lower energy-consumption. Several features are summarised and emphasised in Table 2.1. Finally, all the following candidates excepted 3D XPoint memory can perform logic and arithmetic operations beyond data storage. This feature could be a huge leap in computation acceleration in the field of machine learning and simulations as some matrix-vector applications could be realised in the storage device while the processor executes other parts of the code [31].

3D XPoint memory The first candidate has already been released and was developed by a joint research between Intel and Micron. The technology is named 3D XPoint memory. It promises to deliver fast I/O

	HDD	NAND-Flash	NOR-Flash	DDR-DRAM
Capacity	TBs	TBs	TBs	+100 GB
Read Latency	10 ms	25–50 ns	45 ns	7–15 ns
Write Latency	10 ms	100–500 µs	14 µs	7–15 ns
Endurance	10^{15}	10^{4}–10^{5}	10^{4}–10^{5}	10^{16}
Cost Per GB	$ 0.038	$ 0.05	$ 30–$ 35	$ 1
Density (F^2)	—	4	10	6–10

	PCM	STT-MRAM	ReRAM	Optane™ DC
Capacity	TBs	1 GB	TBs	128–512 GB
Read Latency	60–120 ns	15 ns	10 ns	169–305 ns
Write Latency	50–250 ns	14.5 ns	20–50 ns	94 ns
Endurance	10^{6}–10^{8}	10^{12}	10^{6}–10^{11}	10^{5}
Cost Per GB	$ 1–$ 5	$ 50–$ 100	$ 30–$ 50	$55
Density (F^2)	4–12	6–50	4–10	—

Data used are from [8, 18, 28, 38, 44, 49, 52, 53, 68, 92, 94, 96, 98, 100–102, 121, 123, 137].
Endurance is how many times a memory cell can be rewritten before being worn out (more details in Section 2.1.3.6); higher is better.
Density is the surface required for one memory cell in relative unit (more details in Section 2.1.3.6); lower is better. Intel Optane™ DC performances are considered in the Persistent Memory Module factor.

Table 2.1: Comparison of emerging NVM technologies with DRAM and other storage devices.

access latency and high throughput thanks notably to functionalities that NAND-Flash cannot provide, such as byte-addressability and in-place update. [139] Contrary to NAND-Flash based SSDs, the Optane SSDs can spread I/O over multiple 3D XPoint memories to harness the throughput of multiple memory dice to expose low latency. [62] Byte-addressability and support for data update-in-place solve issues like read-modify-write and write-driven garbage collection that typically exist for conventional SSDs. Both reading and writing performances are on par with NAND-Flash SSDs when using similar internal architecture, or improved performance when using the Optane™ DC SSD media. [139]

PCM The second candidate is PCM (Phase-Change Memory) [137]. This technology is based on using the current to change the crystalline structure of the elements which modifies the resistivity of the component. The material goes from a crystalline state with low resistivity to an amorphous state with high resistivity. By measuring the resistance of the memory cell, it is possible to determine its state. This approach exposes multiple interesting features. First, and similarly to NAND-Flash, it is possible to have a good density of memory cells, to create MLC (Multilevel Cell), and to create 3D-stacked structures [76]. The latency for both reading and writing is also much better compared to Flash, although it may require a higher current to reset a cell's state. Finally the slightly higher price may be counter-balanced by a much greater longevity of the memory cells.

STT-MRAM The third candidate was the STT-MRAM (Spin-Transfer Torque Magnetoresistive Random-Access Memory). Developed by Everspin Technologies and presented in [51], they propose a solution compatible to DDR4 non-volatile, with a latency and an endurance similar to DRAM. However, the capacity is currently very small (256 Mbit with DDR3 in 2018 [51], up to 1 GB DDR4 in 2020 [52]) and the supported frequency (1333 MHz with DDR) seems quite low in comparison to current memory system (1866 MHz to 3200 MHz, from [96], both with DDR) in addition to other deviations from the JEDEC DDR4 specifications. Yet, this technology has been studied by Intel to implement L4 caches into processors [8] to replace eDRAM. They developed scalable macros of 2 MB that could provide at least up to 1 Gbit cache in future architecture. One big limitation nowadays is its price, much higher than any of its competitors.

ReRAM Last, and maybe the most promising candidate in future research, is the ReRAM (Resistive Random-Access Memory), also known as *memristor*. This technology offers the best of the two previous technologies. It is smaller and cheaper than STT-MRAM, with a better performance than PCM with a better endurance. The ReRAM structure is also naturally made for a crosspoint interconnection, allowing for a very high density

factor, and a vertical organisation allowing for 3D-stacking in addition to allow for MLC. [6, 36] This technology has been considered as a replacement for SRAM in caches or as an intermediate body between caches and DRAM.

2.1.4.2.2 How are non-volatile memories used? NVM (Non-Volatile Memory) can serve different purposes in HPC. For its generally substantial capacity, non-volatile memory is often either used for node-oriented storage or network-oriented usage. The SNIA recommended behaviours for software supporting NVM are given in [59]. The technical report on NVM programming model proposes an abstraction and an API for two different levels both as used in the kernel space and their reflection into the user space. The different modes are based on pre-existing abstractions in the system, like memory pages (called *blocks*) or files, to simplify the integration into the kernel.

The NVM.BLOCK mode is at the base of it, as while it is not directly seen by the user, the file system is expected to rely on it for the NVM.FILE mode. Overall the API presented behaves like a transactional database, requiring the respect of the ACID (Atomicity, Consistency, Isolation, Durability) properties. The granularity of data representation is the *block*, which represents a contiguous range of addresses, no matter what means of storage is used.

In NVM.BLOCK mode, as for any kind of memory, a block can be either *mapped*, *unmapped* or *allocated*. When the block is *mapped*, it is allocated on the storage device and in memory and has data written to it. When the block is *unmapped*, the block is on the storage device, but not present in memory. When the block is *allocated*, the memory is allocated to it, but no data have been written back to the storage device yet. Two different modes are proposed for the system abstraction.

In NVM.PM.VOLUME mode, the system is expected to manage the mapping and the synchronisation for the user. However, the NVM.PM.FILE mode allows for the user to access directly the pages mapped to the device (via the MMU translation) with load/store operations, but in this

case the responsibility of synchronisation is left to the end user.

On node storage The first usage for such a device is local storage. The latency to access memory and the bandwidth are critical factors when dealing with a massive amount of data. That is why the locality may be crucial. Having the data *on hand* avoids the cost of network latency and the throttling that can be inflicted to the network traffic.

Usually, in HPC, this storage will be temporary and referred to as scratch space. It can be either implemented with disks on the compute node or on a separated node dedicated for providing *intermediate storage*. The latter solutions are usually provided using NAS (Network-Attached Storage) interfaces. For example IBM Elastic Storage Support [99] uses such technology, based on local disks or using byte-addressable NVDIMM-N based technologies such as Intel Optane™ DC PMM[6] in order to improve both latency and bandwidth.

Finally, and as a pivot case between local-oriented or communication-oriented for large non-volatile memory, some research [61] has found that using close range nodes' scratch space access over the network may also be a sustainable approach. It offers minimum to no overhead, allowing for SSD disks disaggregation which may lead to better resource provisioning and systems being more cost-effective. However this approach is dependent on the network interface capacity to offer DMA, like RoCE or InfiniBand, and requires a compatible NIC to allow for true parallelism and a computation-I/O overlap.

Burst buffer for parallel I/O There is a strong inherent link between network and memory, especially with large amount of memory. This link is growing with the increasing memory requirements. The price of data management on extreme scale computation is still growing both for accessing it in order to do computation but also for exchanging them with other nodes, mitigating network contention. It is also a significant factor as

[6]Technically Optane™ DC PMM is not a NVDIMM-N as it does not comply with JEDEC standard. [68].

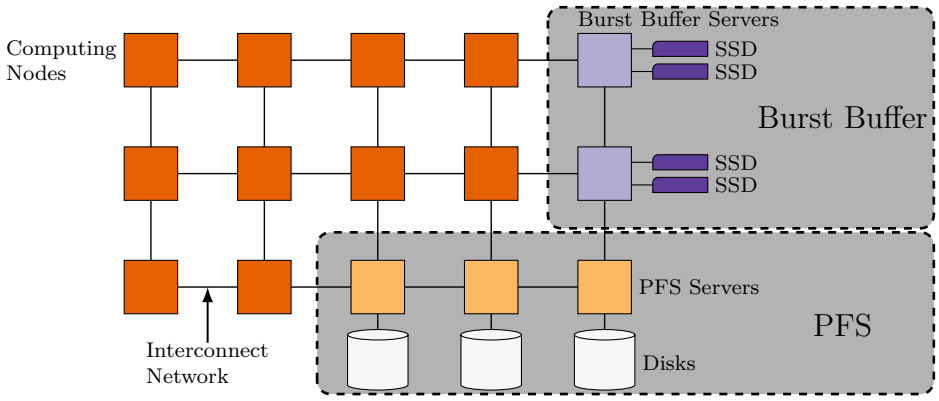


Figure 2.6: Storage nodes dedicated to be used as temporary, intermediate aggregators as used on Cori Supercomputer at NERSC (from [39, Fig. 2]).

the HPC infrastructure relies on PFS which adds reduced pressure to the network.

Many aspects of data retainers used as an intermediate buffer have been studied in the past 5 years. First, they were shown to be useful when used in association with a I/O-aware task scheduler in order to reduce the network bandwidth requirements [63]. Additionally, the details of the multiplicity of memories associated to network topology description and process placement can lead to decreased I/O times in the case of a MIMD application [126]. More straightforward solutions based on these technologies have also been studied for creating close-range intermediate storage node aggregating data for delayed I/O [39] (as shown in Figure 2.6). One final aspect of non-volatile memory being studied is the amount of memory required. The increasing complexity of systems leads to an increased demand in energy and space. It is more important than ever to be able to optimise the requirements to avoid any over or under provisioning. Recent research has explored this question from both an infrastructure provisioning [17] and from a resource allocation point of view [127].

Checkpointing As pointed out by the authors in [53], most long-running HPC applications exceed the MTBF (Mean Time Between Failures) of HPC systems and are therefore vulnerable to hardware or software errors

Systems	Max performance	Checkpoint time (min)
LLNL Zeus	11 TeraFLOP/s	26
LLNL BlueGene/L	500 TeraFLOP/s	20
ANL BlueGene/P	500 TeraFLOP/s	30
LANL RoadRunner	1 PetaFLOP/s	≈20

From [40, Table I]. Source: LLNL, 2010.

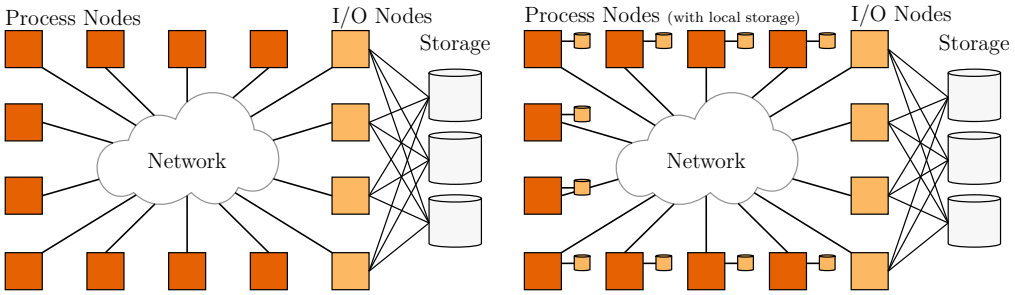
Table 2.2: Time to take a checkpoint on some machines of the TOP500.

during execution. The application has to expose resilience properties to be able to continue their computation after such an event in order to avoid wasting all the time already spent. One solution to prevent this is to make a copy of the current state of the application, across all the nodes involved in the execution. This operation is called checkpointing. In the event of a failure, the application can return to the previously saved state and resume computations from that point onward. This process is called C/R (Checkpoint-Restart). As expressed by Dong et al. in [40] and reported in Table 2.2, in some petascale machine, checkpointing can already take up to thirty minutes and may induce an overhead up to 25 %. One major limitation explained by the author is the limited bandwidth.

Multiple solutions can be used to overcome this restriction. One, exposed in [53], is shifting technology to NVDIMM associated with a disaggregation of storage solutions, as shown in 2.7.

The solution presented is based on Phase-Change Random-Access Memory which is not yet widely available. In addition to having local storage available at each node level, the authors propose a cooperation between close nodes in order to provide higher bandwidth, adding the network bandwidth to the bandwidth of the NVRAM. This technique is used to alleviate the local DRAM bandwidth. Finally, in order to improve the scalability of the solution, the checkpointing technique used is multi-level checkpointing, which can now make better use of the local storage capabilities.

It was expected that by today, PCRAM would have replaced HDDs. Yet, the rapid changes in the underlying technologies mean this outcome never



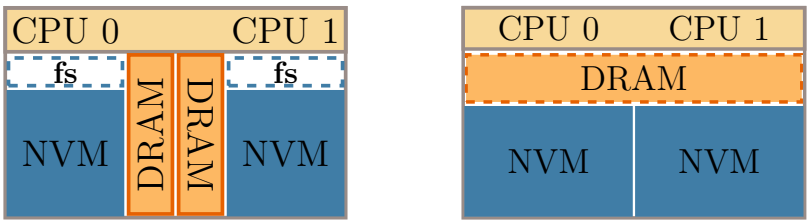
Typical organisation of supercomputers. All the permanent storage devices are controlled by I/O nodes. Disaggregation allows for a new organisation that supports local/global hybrid checkpoint; each process node also has a permanent storage.

Figure 2.7: Disaggregation example, as illustrated by Dong et al. in [40, Fig. 1, Fig. 2].

occurred, since other options became more attractive. The first step was the replacement of spinning hard-drive technology by Flash-based SSDs. The performance has been much improved, although the idea of replacing disks by memory modules connected to working memory is still being developed. Finally, and as of today, it is the 3D XPoint memory technology that has been used to implement the first widely available NVDIMM, with the Intel Optane™ DC PMM.

Working memory NVDIMM provides a much larger capacity for a DRAM-like latency and bandwidth. The form factor of the NVDIMM allows it to be plugged into PCIe slots to ensure optimal performance. Although technically not an NVDIMM as it does not respect the JEDEC specifications for NVDIMM-N or NVDIMM-F and uses a specific protocol (DDR-T), Intel Optane™ DC PMM are the closest products to actual working NVDIMM [68]. In addition to one Optane™ DC PMM, each memory channel must be populated with at least one DRAM DIMM using an address indirection table (AIT) to translate from the DIMM physical address to an internal Optane™ DC media device address.

As presented in [68, 112, 132] and shown in Figure 2.8, Optane™ DC PMM presents two modes: *App Direct* and *Memory mode*. Both modes



(a) App Direct (b) Memory Mode

Dashed borders express layers used transparently for the user.

Figure 2.8: The logical view of configuring all NVDIMMs, as shown in [112, Figure 2].

respect the usages as recommended by [59]. In the former, all the memory is byte-addressable. DRAM and NVM are exposed as shared memory with two NUMA nodes. It is possible to use the persistent memory as if it were main DRAM memory by using the *libvmmalloc* library, which implements SNIA's NVM Programming Model standard [59]. Instead of placing data structures on the system heap, they are placed into a memory mapped file that resides in persistent memory. In the latter mode, only the Optane™ DC PMM is visible as two NUMA nodes to CPUs while DRAM becomes a transparently managed cache, hence, all data objects are placed into NVM.

As presented by Dulloor et al. in [44], data centre applications like key-value stores, in-memory databases, and data analytics are being used to handle exponentially growing datasets but cannot tolerate the performance degradation caused by spilling their workloads to disk. Hence, industry has been studying whether the large capacity of non-volatile memories could be used as an extension to standard DRAM. Recent experiments using Optane™ DC PMM [68] have shown that NVDIMMs are unlikely to be a viable replacement for DRAM on their own.

The industry standard key-value store applications *Memcached* and *Redis* have a penalty of 20.1 % and 23 % respectively when uncached Optane™ DC PMM is used instead of DRAM only. The performance with uncached Optane™ DC is 4.8 % to 12.6 % lower than cached Optane™ DC.

Despite performance losses, Optane™ DC memory allows for far larger sized databases than DRAM due to its density. SPEC 2017 performance benchmark on Optane™ DC shows also that cached mode has similar performance as sole DRAM for integer workloads, but drops by up to 15 % for float workloads.

Peng et al. in [112] also found a higher sensitivity to data locality, as the PMM media is accessed with a 256 bit wide bus, where DRAM is accessed with a 64 bit wide bus. One proposed reason is that the accessing using the internal granularity improves the latency (better prefetching, optimum use of the buffered data) and decreases the write amplification (better longevity for the device).

2.1.5 Summary

As presented, many technologies are cohabiting, each with a specific intended usage. The evolution is driven both by the new usages (e.g. burst buffer, NAS, in-memory checkpointing), and by new technologies (e.g. 3D XPoint memory). However, the combinatorial complexity it creates requires software abstraction to interact with the different memory systems. This eases the testing of different trade-offs without the need to rewrite the program, and, ideally, provides APIs that support the wide variety of solutions, therefore providing portability of code.

2.2 The effect of new memories on programming models

Two approaches are usually followed regarding emerging memories. Contrary to heterogeneous computing systems with embedded memory (GPUs, APUs), it seems that the path followed is an integration with pre-existing data structures and memory models. The main advantage is being able to propose a drop-in solution that works reasonably well performance-wise, without requiring any code modification. However, in order to get even better performance out of the new devices, more precise memory manage-

ment is required. In the following section I will show the already existing APIs for heterogeneous systems, both for heterogeneous computing and heterogeneous memories.

In order to properly understand some mechanisms used when exploiting multi-tier memory, the following section will first introduce the caches and their policies as currently used in processors. Then a short discussion on the possibilities of integrating heterogeneous memory as found in the literature will be presented. Finally, the main frameworks and library dedicated to memory management will be listed and introduced shortly.

2.2.1 Cache policies

In order to provide short latencies, disaggregated memory located close to the core is common in current microarchitectures. These caches have different characteristics, that can change from one level to the next.

Firstly, the caches can be shared between the different cores or be private. Private cores allow for lower latencies as no interconnection is required. In addition, applications running on different cores will have less influence on each other's performance. On the other hand, the shared caches use all the cache space available, even when some cores are idling. It also avoids duplicating shared data. In Figure 2.2, the two lower levels of cache are private for each core, but the L3 is shared between all the cores within the package.

The second characteristic is the cache associativity, or cache placement policy. This determines how to associate blocks of data with cache lines. In a fully associative cache, any block can be stored in any cache line. If no line is free, the cache eviction strategy will be executed to find a victim entry to be evicted from the cache. This provides flexibility and maximises the occupancy of the cache at the cost of time to find an available line and, when none is available, at the cost of executing the cache eviction strategy. At the other end of the associativity spectrum is the direct-map cache. The direct-map associates one memory block to one cache line based on the memory address. This facilitates the cache management as there is no need

to search for an available line nor search for an entry to evict. However this method may lead to a high number of cache conflict misses and a poor occupancy of the cache lines. [110]

Finally, and between the two previously presented strategies, there is the set associative cache. This solution allows for a trade-off between fully-associative and map direct caches. The set size defines the number of cache entries to search from when looking for an available line, and where to look for cache eviction.

In 2020 in Fujistsu's A64FX processors [56], the caches' associativity can be subdivided into sectors in order to reserve entry for one specific data structure, to improve its locality.

Secondly, cache organisation may allow for data redundancy. If full redundancy is ensured between L1 and L2 caches, the cache organisation is said to be *inclusive*. This implies that L1's entries are a subset of L2's entries, but also that any eviction from L2 must be back-invalidated to L1. However the management of cache coherency is simplified as L2 acts as a directory that can simply redirect to L1's corresponding entry. The opposite cache organisation is called *exclusive* and implies that any entry is unique across multiple cache levels. The cache coherency is also simple as well as any entry is unique, however any eviction from L1 will lead to a second eviction from L2 as the evicted line will need to be written back into L2. Also, finding a record requires checking the entries from both L2 and from all L1 caches. There also is a *non-inclusive* cache organisation where new entries to L1 are also added to L2, but eviction from L2 does not trigger a back invalidation from L1.

Finally, multiple eviction strategies have been studied. The objective is to optimise the data reuse and avoid cache misses. Hence the objective is to predict which data will be required next. The decision is usually based on how long since the last time some bytes were accessed and how frequently these bytes are accessed.

Caches on package are usually made using SRAM technologies, but there are alternative approaches, such as using stacked DRAM as a large, high-bandwidth Last Level Cache (e.g. an 'L4' cache), coping with the challenges

of managing the large tag storage required and the relatively slower latencies of DRAM.

2.2.2 Virtual memory spaces for OS-managed multi-tiers memory

As stated by Williams et al. [135], at a high-level, structuring heterogeneous memories as operating system-level NUMA nodes is a natural fit and provides an opportunity to reuse existing OS and application-level abstractions, as shown in Figure 2.2. However, several fundamental differences exist between the homogeneous NUMA and heterogeneous memory systems.

First, heterogeneous memory technologies have significantly different latency and bandwidth, unlike DRAM-based NUMA.

Second, for homogeneous NUMA, the OS-level management aims to increase data locality by increasing CPU access to the data in the local memory socket [72]. This leads to assigning an arbitrary value for the MCDRAM NUMA distance in order to avoid having the system automatically allocating memory there instead of DRAM [70, Chapter 25]. Because of the first two points making the relative weighting to identify the 'bes target" is much more complex.

Third, as noted by Meswani et al. [93], it can be challenging when threads from multiple sockets access the fast-DRAM and desire to allocate there as much memory as possible. The limited amount of memory may not allow every request to be fulfilled at once, especially in many-core contexts, as favouring one thread may be detrimental to another.

To summarise, for heterogeneous memory, although keeping the page granularity and the operating system mechanisms, the challenge is (a) to identify performance-critical data and place them in the fastest memory, (b) to maximise the utilisation of the fast memory with limited capacity, and (c) to share fairly the fast memory across the multiple execution threads [72, 93]. Therefore the management of heterogeneous memory is expected to be much more complicated than homogeneous memory systems and requires either some adaptation of the memory policy management, or a much more

tedious manual management from the application developer.

To add to the already complex environment, Fernando et al. [53] also suggest enriching the heterogeneous environment by considering the memory of close nodes. This enables an extension to the bandwidth offered by the DRAM by envisaging the high-performance network like InfiniBand along with RDMA capabilities.

2.2.3 Programming models for memory

Different programming models are available to manage the plurality of systems. Some programming models do not specifically target the memory heterogeneity (e.g. OpenCL, CUDA, etc.) but the different processing units (e.g. CPU, GPU, etc.). However, co-processors may provide their own embedded memory, for performance reasons, and thus require an interface to exchange data with the main memory. This section will present the different frameworks that are dedicated to managing data across heterogeneous computing systems, then libraries dedicated to data management for heterogeneous memory systems and finally an example of a library level programming model for performance portability with memory management.

2.2.3.1 Frameworks

The frameworks presented are introduced as extensions to C or C++ languages. Apart from OpenMP, their primary objective is to provide an interface between the main memory and the computing devices. On the other hand, OpenMP aims at distributing computations on a single machine with multiple cores, which implies the need for data placement to maintain performance.

2.2.3.1.1 OpenMP The OpenMP API [35] provides a relaxed-consistency, shared-memory model. Each thread is allowed to have its own temporary view of the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. The host device, and target devices that an implementation

Memory space name	Storage selection intent
`omp_default_mem_space`	Represents the system default storage.
`omp_large_cap_mem_space`	Represents storage with large capacity.
`omp_const_mem_space`	Represents storage optimised for variables with constant values.
`omp_high_bw_mem_space`	Represents storage with high bandwidth.
`omp_low_lat_mem_space`	Represents storage with low latency.

From [109, Table 2.8].

Table 2.3: Predefined memory spaces in OpenMP.

may support, have attached storage resources where program variables are stored. The memory will be allocated from the storage resources of the memory space associated with the memory allocator. When an OpenMP memory allocator is not used to allocate memory, OpenMP does not prescribe the storage resource for the allocation; the memory for the variables may be allocated in any storage resource [109]. The standard includes four specific memory spaces in addition to the default one, which are indicated in Table 2.3, and allow the user to provide *hints* to interact with the allocator, for alignment purpose, memory pinning or access rights. The memory allocation can be done across multiple storage resources, and *hints* are also provided to define how the data must be distributed.

2.2.3.1.2 OpenCL The OpenCL standard [74] supports both data-based and task-based parallel programming models. The memory address space is divided into two parts, one being the *host memory*, directly available to the main thread on the host, and the second being the *device memory* which is directly available to kernels executing on OpenCL devices. In addition, the device memory is subdivided into four parts, and compared to OpenMP, more by abstracted characteristics and usage (private memory, local memory, constant memory and global memory) than by physical traits. The allocations are either dynamic (at run-time) or static (defined at compile-time) and reproduced in Table 2.4 The different subdivisions of the device memory are hierarchically staged, similarly to caches with regard

	Global	Constant	Local	Private
Host	Dynamic allocation	Dynamic allocation	Dynamic allocation	No allocation
	Read/Write access to buffers and images but not pipes	Read/Write access	No access	No access
Kernel	Static allocation for program scope variables	Static allocation	Static allocation. Dynamic allocation for child kernel.	Static allocation
	Read/Write Access	Read-only access	Read/Write access. No access to child's local memory.	Read/Write access

From [74, Table 1].

Table 2.4: Memory regions for OpenCL, allocation capabilities and access rights.

to the DRAM. Host memory and device memory may overlap when devices support it, but it requires the implementation to support SVM (Shared Virtual Memory). Finally, the user is responsible for copying data from one device to another, except when the SVM is supported.

2.2.3.1.3 CUDA NVIDIA introduced CUDA [104], a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. CUDA comes with a software environment that allows developers to use C++ as a high-level programming language. At its core are three key abstractions, a hierarchy of

thread groups, shared memories, and barrier synchronisation, that provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. The CUDA programming model, without the *Unified Memory* model presented in the next section, also assumes that both the host and the device maintain their own separate memory spaces in DRAM.

For device-run parts of the applications (namely, the *kernels*), the memory hierarchy is split in three — per-local thread memory, per-block shared memory and global shared memory. There are also two read-only memories that are shared and accessible by all threads, constant and texture memory.

To transfer the data from the host memory to the device, the user has to explicitly call `cudaMemcpy` which may stage the buffer to some pinned memory. In order to optimise memory copies between the host and the device memory, it is possible for the user to pin pages of virtual memory and to share their address with the device addressing space, allowing for just-in-time copies via DMA. These memories are referred to as *zero-copy memories*. The data are still effectively transferred via the PCIe, but not in a single large transaction, and it doesn't need to be coordinated by the user.

2.2.3.1.4 CUDA Unified Memory Unified Memory [103] is a component of the CUDA programming model, first introduced in CUDA 6.0, that defines a managed memory space in which all processors see a single coherent memory image with a common address space. The underlying system manages data access and locality within a CUDA program without the need for explicit memory copy calls. Unified Memory offers a 'single-pointer-to-data' model that is conceptually similar to CUDA's zero-copy memory. The coherency is ensured by explicit synchronisation calls. One key difference between the two is that with zero-copy allocations the physical location of memory is pinned in the CPU system memory so that a program may have fast or slow access to it depending on where it is being accessed from. Unified Memory, on the other hand, decouples memory and execution spaces so that all data accesses are fast.

As far as the host is concerned, no distinction is made in terms of accessing memory allocated with `cudaMallocManaged` or through a `malloc` call. However, although there is no difference in semantics, the programmer needs to be aware that different processors might experience different access times owing to different latencies and bandwidths. Incidentally, a processor is regarded as an independent execution unit with a dedicated MMU; i.e. any GPU or CPU.

2.2.3.2 Memory management libraries

The libraries we shall now present are focused on HPC. They rely on both system memory abstraction (virtual memory, NUMA domains, etc.) and frameworks presented in Section 2.2.3.1.

2.2.3.2.1 AML AML's[7] [114] main focus is to improve data management performance in order to increase computation efficiency. Developed by ANL[8], the objective for this library is to provide a new approach that tackles three memory abstractions: layout, tiling, and movement across a topology. The highly hierarchical memory is only considered as either NUMA nodes or caches, providing data movement between nodes, along with data reshaping. AML supports allocations for HBM, DRAM and GDDR DRAM (only with CUDA). This library also provides an asynchronous mode when multithreading is available. As a building block for improved memory management, this library relies on low level functions to execute the copy, such as `memcpy` or `cudaMemcpy`, which means that the user must detect the system memory topology and provide the parameters for optimal blocking. Finally, the library provides a set of functions to reshape data, extract chunks and move them across memory environments.

2.2.3.2.2 Umpire Umpire[9] [19] is a high-performance, application-oriented, memory allocation library developed by LLNL[10]. This library has

[7] Available at https://xgitlab.cels.anl.gov/argo/aml.
[8] Argonne National Laboratory
[9] Available at https://github.com/LLNL/Umpire.
[10] Lawrence Livermore National Laboratory

a strong software engineering influence with an object-orienting design. It identifies the different memory resources and creates allocators for these. Currently, the library supports HBM, DRAM and GDDR DRAM, both with CUDA and HIP. In addition, although the library is written in C++, it also provides native support for C and Fortran.

The memory can be requested and freed via *allocator* objects whose behaviour may be modified by allocation *strategies*. In order to centralise accesses to Umpire's components, a singleton *ResourceManager* is available. It also provides a simplified portable interface for requesting allocators. The centralisation allows for a precise book-keeping of the memory operations. The export of the logs can be used to replay for analytical purposes or to reproduce problems without the need to run a whole application.

The library shows adaptability, portability and flexibility, however the performance is greatly reduced compared to system allocation for sizes under 256 kB [19, Figure 3].

2.2.3.2.3 SICM SICM[11] [80,134] is a two-level library designed for discovering and managing complex memory hierarchies and sharing resources within them. It has been developed by LANL[12], in collaboration with LLNL[10], ORNL[13] and SNL[14]. The supported memory technologies include HBM, DRAM and NVDIMM. It also supports the different page sizes, and makes it easy to switch between them. The library exposes an interface for C, C++ and Fortran. SICM relies on the *jemalloc* [50] library to provide allocator strategies for different memory tiers, with little fragmentation of memory and fast allocation. The high level is dedicated for automatic optimisations after profiling an execution. In addition, to detect the supported memory environments, the library also provides low level function to detect and measure characteristics like latency, bandwidth and capacity of the different memories. The different memories available are labelled by an enumerate type, and interacted with through the arena allocators.

[11] Available at https://github.com/lanl/SICM.
[12] Los Alamos National Laboratory
[13] Oak Ridge National Laboratory
[14] Sandia National Laboratory

2.2.3.3 Performance portability models with memory support

Some research has also been carried out in order to provide a solution at the crossroads of the two previously introduced levels.

2.2.3.3.1 Kokkos Kokkos[15] [48] is a C++ library for helping the writing of portable performance C++ scientific applications developed and maintained by a cooperation between LLNL[10], ORNL[13] and SNL[14]. It relies on six core abstractions that describe the type of execution support (e.g. device or host), the algorithmic parallelism (e.g. regular loops or tasking), the synchronisation granularity, the memory layer, data layout and access patterns. This approach aims at providing the same portability and adaptability as a language extension such as OpenCL, while staying outside the compilation chain. In addition, the runtime capabilities of this approach enable a dynamic adaptation of data layout during execution. Although providing close to native original performance (within 90 %), using Kokkos is invasive and requires that a significant part of data structures need to be taken over and function markings everywhere, both in the application and the libraries used. Finally, due to the usage of C++ specific construction, no support is possible for languages like C or Fortran.

2.2.3.3.2 oneAPI oneAPI [34] provides a common developer interface across a range of data parallel accelerators. Programmers use DPC++ (Data Parallel C++, a language derived from SYCL) for both API programming and direct programming. oneAPI tries to unify shared memory[16] and distributed memory models into a single API. The final objective of this framework is to provide source-level compatibility, performance transparency and software stack portability.

The application running on the host and the functions running on the devices communicate through memory. oneAPI defines several mechanisms for sharing memory across the platform, depending on the capabilities of

[15] Available at https://github.com/kokkos/kokkos.

[16] Here *shared memory* is considered in the *single compute node*; it is not implied that the memory is necessarily shared across the heterogeneous devices.

the devices. The first one uses *buffer objects* to explicitly copy from host to device or across devices. The second one uses *unified addressing* which ensures that pointer values in the unified address space will always refer to the same location in memory. Finally, the third mechanism is *unified shared memory* which enables data to be shared through pointers without using buffers and accessors. There are several levels of support for this feature, depending on the capabilities of the underlying device.

2.2.3.3.3 SYCL Since SYCL [75] is a single-source programming model, the memory model affects both the main application and the device-specific part of the program. On the SYCL application, the SYCL runtime will make sure data are available for execution of the kernels. As SYCL is based on OpenCL, the memory model is similar. However, SYCL memory objects can encapsulate multiple underlying OpenCL memory objects together with multiple host memory allocations. This enables the same object to be shared between devices in different contexts or platforms. The concurrency in accessing the different objects is specified by the user and managed by the SYCL Runtime.

2.2.3.3.4 RAJA RAJA[17] [20,64] aims at the same objective as Kokkos, but with a less invasive C++ abstraction layer. This allows for an incremental adoption of the framework which is a major concern for very large code base with long life cycle. RAJA targets loop-level parallelism for C++ applications by relying solely on standard C++11 language features for its external interface and common programming model extensions, such as OpenMP and CUDA, for its implementation. Although, dedicated to provide performance portability, the library has been designed to be agnostic with regards to the means of getting data available in the right memory space.

2.2.3.3.5 ROCm HIP Part of the ROCm software stack [46], HIP provides an extension to C++ using all the language constructs to imple-

[17] Available at http://doi.acm.org/10.1145/3295500.3356159.

ment a highly tuned workload for GPUs. HIP allows coding in a single-source C++ programming language including features such as templates, C++11 lambdas, classes, namespaces, and more to implement highly tuned workload for GPUs. The header files for both AMD ROCm and NVIDIA CUDA platforms are both provided by HIP, which allows for specialisation of the code for further platform-specific optimisations. ROCm also provides a 'hipify' tool that automatically converts source from CUDA to HIP, however developers should expect to do some manual coding and performance tuning work to complete the port. Providing support for both platforms, HIP shares its memory model to the one described for CUDA, with the exception of *managed memory*, leaving the responsibility of data transfer to the user.

2.2.3.4 Compiler assisted memory management

In [73], Khaldi and Chapman presented a solution for data placement without code alteration. They proposed to use a new pass during the compilation process with the LLVM toolchain, called BCDA (Bandwidth-Critical Data Analysis). This pass would evaluate and prioritise variables based on the ratio of memory used over the computations, the latency of operations and the presence of simultaneous versus independent accesses using *opt* (the LLVM optimisation tool) and *Polly* (a memory access and pattern analyser). The data with the highest priority had their allocation code changed to use `memkind(3)` in order to be allocated into HBM. Although showing good performance, this approach limits the availability of the solution as it requires the LLVM tool suite environment. Moreover, it forces a more rigid allocation without providing tool to move data out of memory when not required anymore, not does it consider the temporality of the variable lifespan. Finally, this approach lacks the option for the user to experiment and to manually choose the placement of the data, possibly missing the opportunity of changing the data layout or the algorithm in the research for improved performances.

2.3 Analysis and lines of thoughts

From the above review of the current challenges offered by memory systems in HPC, the reflection was organised around the topics to be presented in Section 2.3.1. Section 2.3.2 will present how the problem was approached.

2.3.1 Analysis

2.3.1.1 Memory detection

From the frameworks and libraries presented in Section 2.2.3, only one offers an abstraction that does not rely on the exposition of memory by the system. They mainly rely on the user's knowledge of NUMA domains for memory location for CPUs.

SICM provides an easy to use explicit interface, defining enumerated types to identify memories by their category (e.g. `SICM_DRAM`, `SICM_KNL_HBM` or `SICM_OPTANE`). The memory selection depends on two user provided parameters. First, the *tag*, which identifies the memory kind, and the size if the memory pages, to simplify the usage of industry standard *hugepages* [33]. However, the detection of each memory kind is very much platform dependant, and so are the tags — Intel KNL's MCDRAM is identified with a different tag from IBM PowerPC HBM while both serve the same purpose. Finally, the library also provides automatic tools for measuring the performance (latency and bandwidth) of the different memories.

Recently an interface was introduced to help deal with heterogeneous systems. *Mix and Match Memories* (*M&MMs*) [85] is an extension to `hwloc(2)` [21], the de facto standard for exposing the locality of hardware resources. In addition to the hierarchy of objects based on inclusion and physical location on a server, it suggests querying the memories in a sorted order with respect to their characteristics (called *attributes*). The sorting can be done on one attribute, but a second one can be provided to break a tie. The library expects to be able to use the *Heterogeneous Memory Attributes Table* (*HMAT*) as introduced in the revision 6.2 of the ACPI specification [54]. Another solution may be the export of perfor-

mance metrics into the topology files after benchmarking when the tables are not available. Such a tool, in association with already existing binding capabilities of `hwloc`, may help to create a general approach to memory selection.

2.3.1.2 Performance portability models

As presented in Section 2.2.3.1, the primary focus of the different portable frameworks revolves around processing units. These are often considered as self-contained in the way that they provide their own internal memory organisation that is loosely integrated with the rest of the main memory system and only connected to it. In general, the frameworks' main objective is to copy the data to the device before the start of the computation, with DMA support if the accelerator supports it. The choice of memory in the device is mainly based on the capacity and the access rules to the memory objects. By comparison, the data placement in the host depends more on the memory characteristics. Usually, from the device point of view, only the DRAM is taken in account.

From the frameworks presented, only OpenMP provides an interface for fine-grain memory placement on the host side. However, as the OpenCL's memory model shows, the access to the data may allow for a specific mapping to different memory regions. The device providers are free to create specific memories with different characteristics depending on the memory region. Similarly, the implementers of the OpenCL framework, or other frameworks based on it, are free to map the regions to the heterogeneous memories on the host side.

2.3.1.3 Memory access pattern classification

As analysed by Dulloor et al. in [44], the way data are accessed influences the optimisations that can be made and dictates the memory attributes that are dominant. Microprocessors incorporate prefetchers that locate striding accesses in the stream of addresses originating from execution and prefetch ahead the data in order to reduce the number of cache misses which would

penalise the application performance. Hence analysing the access patterns is important for placing efficiently the data in multi-tiers memory. The first type of memory access is well managed by the prefetcher which can issue data request in advance. The accesses are sequential, independent and regularly strided; this pattern is called *streaming*. When data accessed are still independent but irregular and uniformly distributed across the allocated memory, the pattern is called *random*. Although prefetching is not possible in this *random* case, memory requests can still be grouped in order to issue multiple simultaneous requests. The final pattern is *pointer chasing*. In this case, the processor issues random accesses, but the address for each access depends on the loaded value of the previous one, forcing the processor to stall until the previous load is complete. In this case, the latency cost is the highest, and cannot be reduced unless the data are located in the best latency effective tier.

In addition to pattern evaluation, some tools are dedicated to providing memory object analysis. RTHMS [113] is a memory analysis and advisor that is designed to provide the developer with hints on where to allocate memory. It uses Intel PIN [91] to collect metrics about how the memory is accessed, when in the execution workflow and how frequently. With these statistics and the information about the system, the library advises the programmer with consideration for the limited size of each memory and the interplay between them. This approach provides post-mortem information in order to improve later runs which can be very helpful when tuning an application. The emphasis on the temporal locality in addition to the memory location allows for more complex optimisations and data movement.

Similarly, CoMerge [42] extends the analysis with the utilisation of different memory tiers by multiple applications collocated on the same computational node. This tool provides recommendations for heterogeneous memory sharing between applications in order to avoid seeing critical data moved out of memory. It introduces the notion of the *co-benefit factor* of an application which provides a means of selecting the memory to be prioritised into fast or slow memory while limiting the performance penalty.

2.3.2 Proposed solutions

Although many proposals have been suggested to resolve this matter, none propose a real abstracted view of the system's memory. Describing and addressing memory is very useful, but not sufficient for portability. Some tools like *M&MMs* may help by having a code self adapt when being run on a new platform, without the need to reanalyse the access patterns or change the numerous allocations to explicitly use new memory technologies as they become available. Describing memory requirements in terms of prioritised attributes may be more suitable for increasingly heterogeneous systems where simple hierarchy does not reliably describe the increasing differences in characteristics (i.e. latency, bandwidth, volatility, etc.). Moreover, it enables reusing highly specialised pre-existing libraries that provide partial support for heterogeneous memory systems.

Table 2.5 summarises the different level of support available with the different frameworks, programming models and libraries. A '**?**' marks cases where no data were available, or when the answer is entirely implementation dependent and it is not possible to provide a single definitive answer. As shown, most of the frameworks and programming models do not provide ways of mitigating the allocation cost or reuse allocated data as the allocation is offloaded to high-level APIs. Many recent programming models are based on C++ which limits the availability of other languages despite their broad use in the high-performance computing industry.

Additionally, a low level library approach was preferred. A library seems the option that provides the best integration into users' chain of compilation, as it would not rely on any specific compiler or compiler version although the explicit management of the data layout may be more tedious. Moreover, libraries are less likely to inhibit compiler optimisations.

Finally, the objective was to provide a solution generic enough to be used for application written in C, C++, Fortran or Python. Hence it felt natural to develop it in C as its wrapping would be straightforward for most languages and would work across platforms.

		CPUs			GPUs		Mitigation Strategies	Data Layout	Languages		
		DRAM	HBM	NVDIMM	NVIDIA	AMD			C	C++	Fortran
Framework	CUDA	✓	✗	✗	✓	✗	?	✓	✓	✓	✓
	ROCm	✓	✗	✗	✗	✓	?	✓	✓	✓	✓
	OpenACC	✓	✗	✗	✓	✓	?	✓	✓	✓	✓
	OpenCL	✓	✗	✗	✓	✓	?	✗	✓	✓	✗[†]
	OpenMP	✓	✓[*]	✓[*]	✓	✓	✓	✓[§]	✓	✓	✓
Programming Model	DPC++	✓	✗	✗	✓	✓	?	✓	✗	✓	✗
	HIP	✓	✗	✓	✓	✓	?	✓	✗	✓	✗[†]
	Kokkos	✓	✓	✓	✓	✓	?	✓	✗	✓	✗[†]
	oneAPI	✓	✓	✗	✓	✓	?	✓	✗	✓	✗[‡]
	RAJA	✓	✗	✗	✓	✓	?	✓	✗	✓	✗
	SYCL	✓	✗	✗	✓	✓	?	✓	✗	✓	✗[†]
Libraries	AML	✓	✓	✗	✓	✗	✗	✓	✓	✓	✗
	jemalloc	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓
	libnuma	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
	memkind	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗
	SICM	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗
	tcmalloc	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓
	Umpire	✓	✗	✗	✓	✓	✓	✗	✓	✓	✓

[†] Limited support available via wrapping libraries. [‡] Compiler support in progress.
[*] Support added in OpenMP with version 5.0 [120] to support allocation into memories described by traits. [§] Support added in OpenMP 5.1.

Table 2.5: Memory management libraries and memory support.

2.3.3 Summary and opening

Memory systems are complex and diverse, and using them efficiently requires an understanding of their characteristics and the specifics of the data manipulated. On the other hand, this diversity create new issues that need to resolve. Not all systems are supported by the current software solutions, and the emergence of memories often requires modifications to the code. Moreover, the plurality of memories implies a increased complexity of data placing and data tracking. Data placing is of utmost importance when it comes to performance, as the capacity to select and move the data to the best memory systems avoids starvation and stalling. The following chapter presents a study of a certain class of regular data distributions, and presents a new algorithm that aims to improve data redistribution between two regular distributions.

Chapter 3

Redistributing data between grids with ASPEN

Contents

Efficient use of heterogeneous memory requires data placement management across the different memory tiers. The preliminary work to address this issue was the study of regular data patterns and their distribution across multiple agents. This led to the question of how to compute minimal data transfer to exchange information in order to redistribute data.

HPC applications and libraries have frequently moved parallel data from one distribution scheme to another, for reasons of performance. In modern times, a resurgence of interest in this *data redistribution* problem has emerged due to the need to relocate data distributed across one Producer grid onto a different distribution scheme across a Consumer grid. In this chapter, the efficiency of algorithms used to perform redistribution is studied, and it is shown how the best methods from the literature are still dependent on the number of processors in both grids. A new algorithm ASPEN is described, developed jointly with Adrian Tate. It exploits cyclic patterns and relations in the distribution better, is not dependent on the total number of processors and is thus well suited for use in a workflow management system. This chapter presents a preliminary implementation of the algorithm within such a workflow system and shows performance results that indicate a significant performance benefit in data redistribution generation. The material presented in this chapter is based on the work that led to the publication of the article [55], enriched with the remarks and questions of the community that attended the presentation and engaged afterwards at Euro-Par 2018.

3.1 Introduction

Explicit data movement libraries and tools are used in HPC applications, coupled models, ensembles and workflows, to communicate data between distinct applications through various means. In many HPC workflows, a simulation running on M nodes (the *Producer Grid* or *Producer*) writes a large amount of data to another job running on a possibly distinct set of resources (the *Consumer Grid* or *Consumer*). Although this data movement pattern is far from new, it has become a common concern in modern times

due to the prevalence of data-intensive workflows, coupled climate/environment applications and combined workflows of HPC with data analytics or AI. Many approaches exist to provide data movement between programs including in-situ frameworks, job couplers, in-memory databases and file-system approaches. In this chapter I describe a library for communicating data between jobs over the interconnect fabric. Moving data over the interconnect, directly from memory has the benefit that many fewer data copies are incurred, but has the significant hurdle of needing to explicitly manage the parallel data movement in order to move the data. This problem of explicit data redistribution management is the focus of this chapter.

A good deal of work (reviewed in Section 3.3) has explored the cost and benefits of explicit data redistribution, typically to a different distribution scheme within *the same processor grid*. While sharing many qualities with the classical data redistribution problem, so called *Producer-Consumer redistribution* or *M:N node redistribution* exhibits significant additional complications arising from the fact that the two grids reside in different jobs and lack awareness of the other's characteristics including distribution scheme. Cray has developed a library called the *Universal Data Junction* (UDJ)[1] that provides the missing information and allows distinct jobs in distinct grids to package, send and receive parallel distributed data over the high-performance interconnect as well as other resources that may be preferred.

In this work, the focus is on the algorithmic machinery that is required in order to allow a Producer and a Consumer Grid to communicate the correct data, at minimal expense in a scalable fashion. The reason for placing so much emphasis on the cost of redistribution is that the operations cannot easily be offloaded or performed asynchronously and thus incur direct overhead on the simulation code, which is often intolerable. The redistribution process can be decomposed into three distinct parts: (1) indexation, which compute the index and size of blocks of data to be exchanged; (2) packing (unpacking), which copies the data to (from) the exchanged buffer; and (3) communication, which convey the buffer from the producer to the consumer. The benefice for the whole communication of improving the first

[1] https://gitlab.com/cerl/universal-data-junction

part is hard to evaluate, as the three part are orthogonal to each other. Mainly, the packing and communication costs are very closely related to the overall amount of data that needs to be exchanged, while it is not necessarily the case for the indexation operation[2]. As shown in [65, Tables 2, 3 and 5], the indexation time is not scaling with the amount of data to index. However, the amount of data to be moved is a strong factor of the packing and communication time, which is limited by the memory bandwidth and the network characteristics. For example, in [65, Table 3], for a 1-d BLOCK-CYCLIC of size 500, over 8 producers to a 1-d BLOCK-CYCLIC of size 3 over 5 consumers, for 80 000 elements to data, the indexation time is the biggest contribution to the total time (4.3 ms indexation, 3.1 ms packing, 2.9 ms communication); on the other hand, for 20 000 000 elements, the packing time is 1395 ms and the communication time is 800 ms. Hence, for one given redistribution scheme, the indexation fluctuated from 40 % to 0.14 % of the total redistribution time. Yet, the improved performance of memory and network may lead to an increased influence of the indexation time to the total redistribution operation. As shown in the different equations in Section 3.4, the time complexity of the redistribution has multiple factors, including the number of producers, number of consumers, and the relative size of the two BLOCK-CYCLIC distributions. While this indexation may be cached, two major applications benefiting from an improved redistribution algorithm are the resiliency over defective nodes which require a load balancing with a redistribution of data, and an elastic scaling of resources during execution. Both these applications require to redo the indexing computation.

In Section 3.3, it is shown that classical algorithms and those in the literature display running times that are proportional to the number of *remote* processors from the perspective of either the Producer (i.e. *remote* means Consumer) or the Consumer (i.e. *remote* means Producer) grid. In the exascale era, it is expected that simulation jobs may run on millions of compute cores. Hence, this dependence on remote grid size may become intolerable.

[2]See Section 3.4 for the complexity of each algorithm.

In Section 3.4 a new approach is described. It exploits three types of periodicity in cyclic data distributions, resulting in a lower complexity redistribution algorithm and one that does not depend on remote grid sizes. Section 3.5 shows the results of this new approach versus the classical algorithm and some of the most used and well-regarded algorithms published in the literature.

3.2 Background

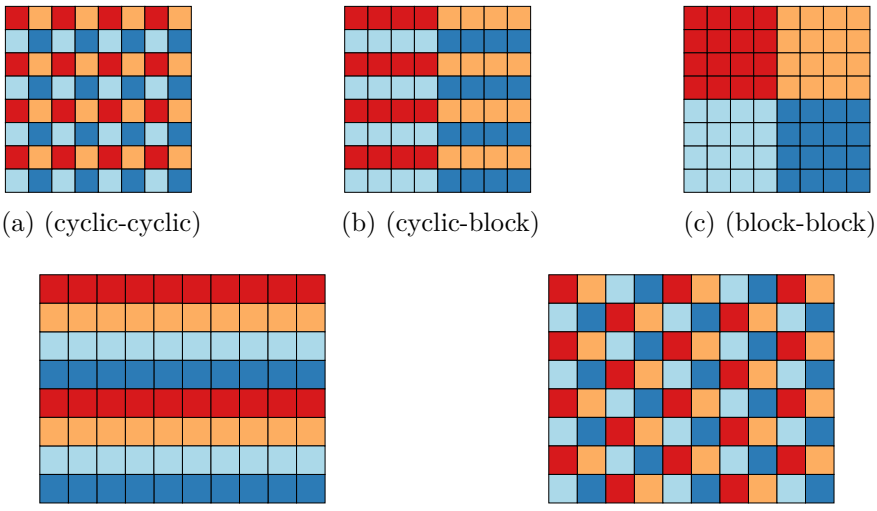
For this chapter, the regular redistribution problem is defined in the same way as [60] using updated producer-consumer terminology: given a d-dimensional array A on a set of Producer resources (processors and memory) $\mathscr{R}_{producer}$ that uses some distribution scheme $\mathscr{D}_{producer}$ we wish to move all the data to another set of resources $\mathscr{R}_{consumer}$ using some other distribution scheme $\mathscr{D}_{consumer}$. $\mathscr{D}_{producer}$ and $\mathscr{D}_{consumer}$ represent arbitrary array element mappings across each dimension of the array. The global array indices of A are given by $G^1, \ldots, G^d$. The set of distribution schemes of primary interest are BLOCK, CYCLIC and BLOCK-CYCLIC, as presented in [37] and shown in Figure 3.1. Since k-d BLOCK-CYCLIC is a generalisation of BLOCK, CYCLIC and 1-d CYCLIC, only the former case is studied in this chapter. The different combinations shown in Figure 3.2 are obtained from *lifting* [37]: (1) by applying a distribution per dimension (e.g. Figures 3.2-a to 3.2-c); (2) by applying a distribution per dimension but ignoring one (or more) of the dimensions, e.g. in Figure 3.2-d the column-dimension is ignored (cf. *) and the rows are attributed in a cyclic (round-robin) manner; or (3) by applying a distribution to a logically flattened data structure, e.g. Figure 3.2-e shows a (cyclic)-distribution applied to a 2D-array as if it was a 1D-array with a row major data layout.

Like [60] and [119], the LDD (Local Data Descriptor) approach is used, but its representation was ignored since it is an implementation feature not relevant to the algorithmic descriptions. Local data sizes of A on rank p are given by $L_1^p, L_2^p, \ldots, L_d^p$. Processors compose a d-dimensional processor grid $p_1 \times \cdots \times p_d$ where $p_i, (1 \leq i \leq d)$ gives the number of processors in the grid

(a) (cyclic)-distribution (b) (block-cyclic)-distribution (c) (block)-distribution

From [37, Fig. 3], the colour scheme has been changed for better readability.

Figure 3.1: The three common distributions of a one-dimensional regular data structure over four places.



(a) (cyclic-cyclic) (b) (cyclic-block) (c) (block-block)

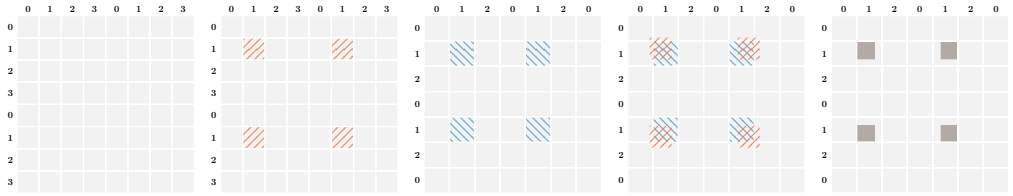
(d) (cyclic-*) (e) (cyclic)

From [37, Fig. 4], the colour scheme has been changed for better readability.

Figure 3.2: Various combinations of the common predefined distributions applied to a two-dimensional regular data structure.

dimension i. The case study discusses two such processor grids $\mathscr{G}_{producer}$ and $\mathscr{G}_{consumer}$ where the resources are assumed to be distinct though this is not necessary. The mapping $\texttt{G2L}(p, d)$ is defined as the function that maps global indices to the local indices for processor p in dimension d, and the inverse relation $\texttt{L2G}(p, d)$ mapping local indices to global indices.

The non-triviality of redistribution of cyclic data can be illustrated by the graphic example of Figure 3.3. A 2-d array is divided into 2-d partitions using some block sizes b_1^1, b_2^1. The blocks of this partitioning are distributed using a 2d block-cyclic distribution scheme across a producer grid of size 4×4. The block ownership is denoted by labelling the blocks by processor owner, round-robin style along each dimension (Figure 3.3-a). The objective



(a) (b) (c) (d) (e)

Figure 3.3: Example of non-triviality of data index calculations for trivial distribution across 4×4 producer and 3×3 consumer grids with different block sizes.

is to redistribute the same 2-d data across a different consumer grid of size 3×3 using different block sizes b_1^2, b_2^2 labelled similarly (Figure 3.3-b). For any process pair (p, c) where p is in the producer grid and c is the consumer grid, we can overlay the global data owned by each processor to begin to ascertain shared indices, e.g. producer process $(0, 0)$ (Figure 3.3-b) and consumer process $(0, 0)$ (Figure 3.3-c) superimposed in Figure 3.3-d. The intersection of the superimposed data in Figure 3.3-e represents the global indices that these two processors must directly exchange over the network (i.e. the producer $(0, 0)$ must send these indices and the consumer $(0, 0)$ must receive these indices). The d-dimensional situation is a direct extension of the illustrated 2-dimensional case.

3.3 Related Work

The question of parallel data redistribution has been addressed many times, both statically and dynamically as this question was central when dealing with the imposed data distributions of early distributed memory programming models such as HPF (High-Performance Fortran) [90, 117]. Extensive analysis has been performed on both the nature of block-cyclic distribution, and its relevance to distributed memory relations as it stands as a generalisation of both block distribution and cyclic distribution. Multiple improvements have been proposed taking advantage of certain characteristics of this kind of data distribution [60, 65, 117, 119]. These solutions

also focused on the message scheduling part of array redistribution, which is beyond the scope of this chapter. Petitet and Dongarra [115] described techniques for redistribution taking into account the severe alignment restrictions induced by the architecture, as well as further treatment of the scheduling.

Thakur et al. compared different solutions and presented both specific and general solutions varying the size of the blocks but restricted to fixed size process grids [128, 129]. The techniques presented rely on computing the source and destination for each element of the array outside of where it was possible to use improvements due to any common factors between the two block-cyclic sizes. Ching-Hsien Hsu et al. [66] also described some optimisations for specific cases where the two block-cyclic distributions have a common factor, but with an irregular number of processors. In their more generic approach [65] the authors provide a thorough proof of the algorithm. The algorithm presented in this chapter, although very close in principle, is based on LDD (Local Data Descriptor) usage and does not enforce the sizes to be relatively prime numbers, allowing simpler generation of the final scalar product.

3.4 Data Redistribution Algorithms

From the literature, the redistribution problem has been expressed as: given two possibly different regular distributions of data over two grids P and C with distributions D_P and D_C, for each pair of processes $(p \in P, c \in C)$ find the intersecting elements in $D_p \cap D_c$. The general idea when addressing the redistribution problem is to compute the intersecting `blocks` of data between the ranks. Each block is characterised by its starting index, its length, its dimension and its destination. As stated in [119], a multidimensional distribution can be expressed as a cross-product of multiple one-dimensional distributions. Using this approach, the general solution for a multidimensional approach is presented in Section 3.4.1, while Sections 3.4.2 to 3.4.5 focus more specifically on the required block comparisons.

3.4.1 General Problem

Algorithm 1 Base algorithm for redistribution

▷ Each rank performs the following operations

Input

P	Producer Grid
C	Consumer Grid
D_P	Producer Distribution
D_C	Consumer Distribution

Output

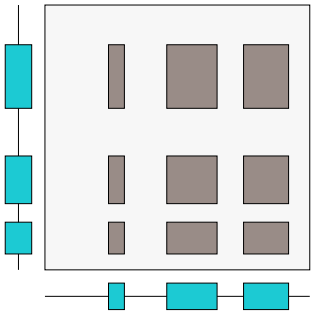
I_{ranks}	Set of $\texttt{blocks}^{\texttt{d}}$ (set of tuples)

1: **for** $\texttt{d} \leftarrow 1$ **to** $\texttt{ndims}(Data)$ **do**
2: $\quad \texttt{blocks}^{\texttt{d}} \leftarrow \texttt{ComputeIntersection}(P^{\texttt{d}}, C^{\texttt{d}}, D_P^{\texttt{d}}, D_C^{\texttt{d}})$
3: $\quad I_{rank}^{\texttt{d}} \leftarrow I_{rank}^{\texttt{d}} \cup \texttt{blocks}^{\texttt{d}}$
4: **end for**

Algorithm 1 presents the outer loop over the dimensions of the array. This algorithm generates the sets of blocks describing how to scatter the local data. In this algorithm, **`ComputeIntersection`** refers to any of the algorithms presented in Sections 3.4.2, 3.4.3, 3.4.4 and 3.4.5. The version described here considers the computation of the complete redistribution. It is however possible to pass the remote process coordinates to Algorithms 2 or 3 in order to compute the unique intersection with the local process. As the full description of the intersecting blocks is created with a crossproduct, any empty returned $\texttt{block}^{\texttt{d}}$ would allow the algorithm to finish early in this case. The final objective is to be able to generate the coordinates of the blocks of consecutive data that belong to the grey areas of the matrix represented in Figure 3.4.

3.4.2 Classical Algorithm

This algorithm presents the naïve way of computing the intersection, by taking each block of the given local distribution and looking for an overlap by comparing its boundaries with those of each block of the remote distribution. This comparison is performed for each process of the remote grid.



Inspired by [117, Figure 5].

Figure 3.4: Intersection figure of two 1-d data redistribution patterns.

Algorithm 2 Classical Redistribution Algorithm.

▷ Each rank performs the following for each dimension `d`

Input

P^d	Producer Grid
C^d	Consumer Grid
D_P^d	Producer Distribution
D_C^d	Consumer Distribution

Output

$I_{rank}^{\mathtt{d}}$	Set of tuples of the form (`remoteRank`, `start`, `end`)

$N_{local} \leftarrow$ Number of local blocks owned by this rank
for `remote` $\leftarrow 1$ **to** $|C^d|$ **do**
  $N_{\mathtt{remote}} \leftarrow$ Number of local blocks on remote rank
  for `localBlockId` $\leftarrow 1$ **to** N_{local} **do**
    `localBlock` $\leftarrow$ **`getBlock`**$(D_P^d, \mathtt{localBlockId})$
    for `remoteBlockId` $\leftarrow 1$ **to** N_{remote} **do**
      `remoteBlock` $\leftarrow$ **`getBlock`**$(D_C^d, \mathtt{remoteBlockId})$
      `Left` $\leftarrow \max(\mathtt{localBlock}.start, \mathtt{remoteBlock}.start)$
      `Right` $\leftarrow \min(\mathtt{localBlock}.end, \mathtt{remoteBlock}.end)$
      if `Left` $<$ `Right` **then**
        $I_{rank}^{\mathtt{d}} \leftarrow I_{rank}^{\mathtt{d}} \cup (\mathtt{remote}, \mathtt{Left}, \mathtt{Right})$
      end if
    end for
  end for
end for

The total number of operations is given by

$$\texttt{Ops}_{Classical} = D \cdot L \cdot R \cdot N_{local} \cdot N_{remote} \tag{3.1}$$

where N_{local} represents the number of local blocks and L represents the number of processes in the local grid dimension, respectively remote blocks and remote grid dimensions are N_{remote} and R.

3.4.3 FALLS Algorithm

This algorithm [118] is one of the best versions found in the literature for *M:N node redistribution*. The same idea is expressed in [60, 119], and summarised in Algorithm 3. Although comparing boundaries block-by-block, these results present a huge improvement over the classical algorithm in terms of the number of block comparisons required.

The bounds are reduced by using the fact that the intersection of two block cyclic distributions can be expressed as the union of some sets of block cyclic distributions, each origin being the beginning of the intersection, each block length being the length of the intersecting block and the distance between blocks[3] being equal to the lower common multiple of the two original strides. The result is that it is only necessary to compare blocks within one stride S. In other words, for each element x in the found intersection then $x + n(S)$ is also inside the intersection, where n is all integers for which $x + n(S)$ remains smaller than the extent of the full array. Additionally, it is only necessary to compare the blocks in the onwards direction and those already checked can be ignored.

Compared to the classical algorithm, the reduction of the bounds drastically reduces the number of blocks to be considered when evaluating the intersection for big grids of data. The total number of operations is given by

$$\texttt{Ops}_{\text{FALLS}} = D \cdot L \cdot R \cdot \hat{N}_{local} \cdot \hat{N}_{remote} \tag{3.2}$$

where the $\hat{N}_{local}$ and $\hat{N}_{remote}$ represent the reduced number of local blocks

[3]Later referred to simply as stride.

Algorithm 3 FALLS Redistribution Algorithm.

▷ Each rank performs the following for each dimension `d`

Input

P^d	Producer Grid
C^d	Consumer Grid
D_P^d	Producer Distribution
D_C^d	Consumer Distribution

Output

$I_{rank}^{\mathtt{d}}$	Set to tuples of the form (`remoteRank`, `start`, `end`)

1: $S_{local} \leftarrow$ local stride between blocks
2: $S_{remote} \leftarrow$ remote stride between blocks
3: $\mathtt{S} \leftarrow \mathtt{LCM}(S_{local}, S_{remote})$
4: $N_{local} \leftarrow$ Number of local blocks owned by this rank
5: **for** $\mathtt{remote} \leftarrow 1$ **to** $|C^d|$ **do**
6: $\quad N_{remote} \leftarrow$ Number of local blocks on remote rank
7: $\quad$ **for** $\mathtt{localBlockId} \leftarrow 1$ **to** $\max(N_{local}, \frac{\mathtt{S}}{S_{local}})$ **do**
8: $\quad\quad \mathtt{localBlock} \leftarrow \mathtt{getBlock}(D_P^d, \mathtt{localBlockId})$
9: $\quad\quad \mathtt{firstIndex} \leftarrow \max(0, \lceil \frac{\mathtt{localBlock}.start - \mathtt{remoteOffset} - \mathtt{remoteBlksz}}{S_{remote}} \rceil)$
10: $\quad\quad \mathtt{lastIndex} \leftarrow$
$\quad\quad\quad \min(1 + \frac{\mathtt{localBlock}.start + \mathtt{localBlocksize} - \mathtt{remoteOffset}}{S_{remote}}, N_{remote}, \frac{\mathtt{S}}{S_{remote}})$
11: $\quad\quad$ **for** $\mathtt{remoteBlockId} \leftarrow \mathtt{firstIndex}$ **to** $\mathtt{lastIndex}$ **do**
12: $\quad\quad\quad \mathtt{remoteBlock} \leftarrow \mathtt{getBlock}(D_C^d, \mathtt{remoteBlockId})$
13: $\quad\quad\quad \mathtt{Left} \leftarrow \max(\mathtt{localBlock}.start, \mathtt{remoteBlock}.start)$
14: $\quad\quad\quad \mathtt{Right} \leftarrow \min(\mathtt{localBlock}.end, \mathtt{remoteBlock}.end)$
15: $\quad\quad\quad$ **if** $\mathtt{Left} < \mathtt{Right}$ **then**
16: $\quad\quad\quad\quad$ **for** $\mathtt{disp} \leftarrow 0$ **to** $\frac{|Data^d|}{\mathtt{S}}$ **do**
17: $\quad\quad\quad\quad\quad \mathtt{start} \leftarrow \mathtt{Left} + \mathtt{disp} \times \mathtt{S}$
18: $\quad\quad\quad\quad\quad \mathtt{end} \leftarrow \mathtt{Right} + \mathtt{disp} \times \mathtt{S}$
19: $\quad\quad\quad\quad\quad I_{rank}^{\mathtt{d}} \leftarrow I_{rank}^{\mathtt{d}} \cup (\mathtt{remote}, \mathtt{start}, \mathtt{end})$
20: $\quad\quad\quad\quad$ **end for**
21: $\quad\quad\quad$ **end if**
22: $\quad\quad$ **end for**
23: $\quad$ **end for**
24: **end for**

due to searching only with one S. In theory then, Equation 3.2 resembles Equation 3.1 but in practice, $\hat{N}$ and N typically differ greatly with $\hat{N} \ll N$.

3.4.4 ScaLAPACK

Algorithm 4 describes the implementation of the redistribution algorithm as proposed by the ScaLAPACK [32] library and presented in [117]. This algorithm also bases its reduction of complexity by using the periodic structure of the block-cyclic data-sets intersection. In [117], the authors also prove the maximum bounds of the number of elements in the intersection is at most $\texttt{LCM}(S_{local}, S_{remote})$, henceforth reducing the intersection computation to every element within the minimum between one stride S (as expressed in 3.4.3), and the full extent of the array in that dimension. This algorithm goes through all the elements within one stride, incrementing progressively by block as in a merge part of a merge-sort algorithm in order to determine overlap areas. The complexity of this algorithm is similar to the complexity expressed in Equation 3.2. However, the number of comparisons is marginally improved as the crawling always goes forward, contrary to the FALLS algorithm, without the onward only comparison.

3.4.5 ASPEN Algorithm Description

To improve redistribution performance, a scheme can be developed, that would exploit further qualities of the periodic nature of the distributed data, and the known relationships between adjacent blocks. This approach is called *Adjacent Shifting of PEriodic Node data* (*ASPEN*). To illustrate the approach the following sections will describe the two remaining weaknesses of the existing algorithms.

3.4.5.1 Periodicity of Remote Block Data

In the Algorithms 2 and 3 each local block's position in the global scheme is compared against multiple remote blocks (all remote blocks in the case of Algorithm 2 and many fewer than all remote blocks in the case of 3).

Algorithm 4 ScaLAPACK Redistribution Algorithm.

▷ Each rank performs the following for each dimension `d`

Input

P^d	Producer Grid
C^d	Consumer Grid
D_P^d	Producer Distribution
D_C^d	Consumer Distribution

Output

$I_{rank}^{\mathtt{d}}$	Set to tuples of the form (`remoteRank`, `start`, `end`)

1: $S_{local} \leftarrow$ local stride between blocks
2: $S_{remote} \leftarrow$ remote stride between blocks
3: $\mathtt{S} \leftarrow \min(\mathbf{LCM}(S_{local}, S_{remote}), |Data^d|)$
4: **for** $\mathtt{remote} \leftarrow 1$ **to** $|C^d|$ **do**
5: `localBlockId` $\leftarrow 1$
6: `localBlock` $\leftarrow \mathbf{getBlock}(D_P^d, 1)$
7: `remoteBlockId` $\leftarrow 1$
8: `remoteBlock` $\leftarrow \mathbf{getBlock}(D_P^d, 1)$
9: **while** `localBlock`.$end \leq \mathtt{S} \wedge$ `remoteBlock`.$end \leq \mathtt{S}$ **do**
10: `Left` $\leftarrow \max($`localBlock`.$start$, `remoteBlock`.$start)$
11: `Right` $\leftarrow \min($`localBlock`.end, `remoteBlock`.$end)$
12: **if** `Left` $<$ `Right` **then**
13: **for** `disp` $\leftarrow 0$ **to** $\frac{|Data^d|}{\mathtt{S}}$ **do**
14: `start` $\leftarrow$ `Left` $+$ `disp` $\times$ `S`
15: `end` $\leftarrow$ `Right` $+$ `disp` $\times$ `S`
16: $I_{rank}^{\mathtt{d}} \leftarrow I_{rank}^{\mathtt{d}} \cup (\mathtt{remote}, \mathtt{start}, \mathtt{end})$
17: **end for**
18: **end if**
19: **if** `Left` $=$ `localBlock`.$start$ **then**
20: `localBlockId` $\leftarrow$ `localBlockId` $+ 1$
21: `localBlock` $\leftarrow \mathbf{getBlock}(D_P^d,$ `localBlockId`$)$
22: **end if**
23: **if** `Left` $=$ `remoteBlock`.$start$ **then**
24: `remoteBlockId` $\leftarrow$ `remoteBlockId` $+ 1$
25: `remoteBlock` $\leftarrow \mathbf{getBlock}(D_P^d,$ `remoteBlockId`$)$
26: **end if**
27: **end while**
28: **end for**

Algorithm 5 ASPEN Redistribution Algorithm.

▷ Each rank performs the following for each dimension `d`

Input

P^d	Producer Grid
C^d	Consumer Grid
D_P^d	Producer Distribution
D_C^d	Consumer Distribution

Output

$I_{rank}^{\mathtt{d}}$	Set of tuples of the form (`remoteRank`, `start`, `end`)

1: $S_{local} \leftarrow$ local stride between blocks
2: $S_{remote} \leftarrow$ remote stride between blocks
3: $\mathtt{S} \leftarrow \mathtt{LCM}(S_{local}, S_{remote})$
4: $N_{local} \leftarrow$ Number of local blocks owned by this rank
▷ `GTL` is a global to local index conversion function.
5: **for** $\mathtt{localBlockId} \leftarrow 1$ **to** $\max(N_{local}, \frac{\mathtt{S}}{S_{local}})$ **do**
6: $\quad \mathtt{localBlock} \leftarrow \mathtt{getBlock}(D_P^d, \mathtt{localBlockId})$
7: $\quad \mathtt{remote} \leftarrow \frac{\mathtt{localBlock}.start}{\mathtt{remoteBlksz}} \bmod |C^d|$
8: $\quad \mathtt{offset} \leftarrow \mathtt{localBlock}.start \bmod \mathtt{remoteBlksz}$
9: $\quad \mathtt{Left} \leftarrow \mathtt{localBlock}.start$
10: $\quad \mathtt{nextStartInblock} \leftarrow \mathtt{localBlock}.start - \mathtt{offset} + \mathtt{remoteBlksz}$
11: $\quad$ **if** $\mathtt{nextStartInBlock} \leq \mathtt{localBlock}.end$ **then**
12: $\quad\quad \mathtt{Right} \leftarrow \min(\mathtt{nextStartinBlock}, |Data^d|)$
13: $\quad\quad \mathtt{diff} \leftarrow \mathtt{Right} - \mathtt{Left}$
14: $\quad\quad$ **for** $\mathtt{ps} \leftarrow \mathtt{Left}$ **to** $|Data^d|$ **by** S **do**
15: $\quad\quad\quad \mathtt{start} \leftarrow \mathtt{GTL}(\mathtt{ps})$
16: $\quad\quad\quad \mathtt{end} \leftarrow \mathtt{GTL}(\min(\mathtt{ps} + \mathtt{diff}, |Data^d|))$
17: $\quad\quad\quad I_{rank}^{\mathtt{d}} \leftarrow I_{rank}^{\mathtt{d}} \cup (\mathtt{remote}, \mathtt{start}, \mathtt{end})$
18: $\quad\quad$ **end for**
19: $\quad\quad \mathtt{Left} \leftarrow \mathtt{Right}$
20: $\quad\quad \mathtt{remote} \leftarrow (\mathtt{remote} + 1) \bmod |C^d|$
21: $\quad$ **end if**

▷ See end of the algorithm in Algorithm 5

Algorithm 5 ASPEN Redistribution Algorithm (continued).

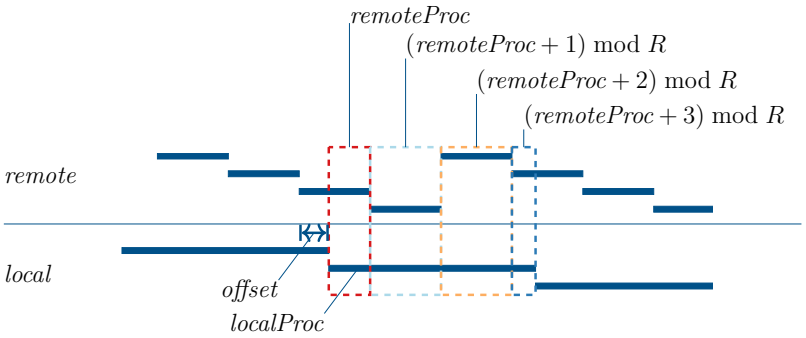
```
22:     for Left to min(localBlock.end, |Data^d|) by remoteBlksz do
23:         Right ← min(Left + remoteBlksz, localBlock.end, |Data^d|)
24:         diff ← Right − Left
25:         for ps ← Left to |Data^d| by S do
26:             start ← GTL(ps)
27:             end ← GTL(min(ps + diff, |Data^d|))
28:             I^d_rank ← I^d_rank ∪ (remote, start, end)
29:         end for
30:         remote ← (remote + 1) mod |C^d|
31:     end for
32:     Right ← min(localBlock.end, |Data^d|)
33:     if Left ≤ Right then
34:         diff ← Right − Left
35:         for ps ← Left to |Data^d| by S do
36:             start ← GTL(ps)
37:             end ← GTL(min(ps + diff, |Data^d|))
38:             I^d_rank ← I^d_rank ∪ (remote, start, end)
39:         end for
40:     end if
41: end for
```

In fact, the need to perform more than one comparison ignores periodic qualities of the data distribution since the constant stride should enable a direct periodic comparison. Consider the code in Algorithm 3 lines 11–14. This code searches over the loop of remote blocks (Algorithm 3 line 11) to generate all remote `RemoteBlockIDs`, then inside that loop `remoteBlock` is extracted using **`getBlock`**$(D_C^d, \texttt{remoteBlockId})$ (Algorithm 3 line 12). `Left` and `Right` are then both generated using various extents of `localBlock` and `remoteBlock` (Algorithm 3 lines 13 and line 14). This logic can be avoided if a *periodic offset* is generated as follows

$$\texttt{offset} \leftarrow \texttt{localBlock}.\mathit{start} \bmod \texttt{remoteBlksz}$$

with `offset` visually represented in Figure 3.5 and appearing in Algorithm 5 line 8.

The `offset` can be used to indirectly obtain the same information, with-



offset is a periodic difference that will mean a local-remote comparison is valid for this local block when *offset* is greater than a threshold. The leftmost part of local data maps to processor *remoteProc*. Adjacent data on the local processor can be known to then map to *remoteProc+1* (and repeated for any further adjacent blocks); R represents the remote grid size in the considered dimension.

Figure 3.5: Illustration of adjacent shifting and periodic relations.

out making explicit comparisons with individual remote blocks, by checking the inequality

$$(\texttt{localBlock}.start - \texttt{offset} + \texttt{remoteBlksz}) \leq \texttt{localBlock}.end \quad (3.3)$$

If condition 3.3 is true, then this particular local and remote block comparison overlaps on the left-hand side of the local block. This can be understood by seeing that the blue box of Figure 3.5 would be non-empty when 3.3 holds. When it holds, $\texttt{remoteBlksz} - \texttt{offset}$ elements will be shared with processor `remote`. This is how ASPEN exploits the periodic nature of remote data to avoid looking at all remote blocks.

3.4.5.2 Properties of adjacent Sub-blocks

In the case that condition 3.3 holds, some number of elements are shared with processor `remote`. Instead of resetting knowledge with respect to the rest of the local block, ASPEN exploits the fact that if a set of global indices $g_l, \ldots, g_r$ of length less than `localBlockSize` map to processor `remote`, and if some sets of global indices $\{g_{r+1}, \ldots, g_{r+p}\}$ with $p + (r - l) \leq \texttt{localBlockSize}$ then `localProc` will also share indices with

processor $(\texttt{remoteProc} + 1) \bmod |C^d|$. Similarly, if several blocks of size `remoteBlksz` fit into the `localBlock`, then each full block will map to the next processor in the remote grid. This approach is how ASPEN assigns contiguous local sub-blocks to adjacent processors in the remote grid (adjacency shifting). Hence the loop over remote processors in Algorithm 2 line 6 and Algorithm 3 line 11, does not appear in 5. The number of operations in Algorithm 5 is given by

$$\texttt{Ops}_{\text{ASPEN}} = D \cdot L \cdot \hat{N}_{local} \tag{3.4}$$

Comparing this to Equation 3.2, a factor of $R \cdot \hat{N}_{remote}$ reduction in operations appears. The missing R term in particular will affect scalability since each grid will not require distinct calculations for every process element in the size of the remote grid. Theoretically then, ASPEN is expected to scale significantly better with larger Producer or Consumer grids involved in redistribution.

3.5 Results

The following tests were run on Cray XC30 systems, each node featuring two Intel Xeon Haswell E5–2698 with 16 cores each (2.30 GHz). The benchmark was made of MPI applications independently computing the complete redistribution from one 2D grid of processes to another 2D grid, varying the dimensions, shape and size of each grid. The data was a fixed size 2D square grid of ten thousand by ten thousand elements. The size was chosen arbitrarily such as it allows having at least one full stride, with different block sizes (up-to 256), with a one-dimension grid of 32 processes. Because this benchmark aimed at evaluating the redistribution performances, the computations were only executed on the indices and no actual communication of data occurred. In addition, changing the total size of data only penalises the *classical* naïve approach, and would artificially improve the performance gain of all 4 better optimised solutions.

Each case of block-cyclic to block-cyclic distribution was run at least 20

times for all 5 methods: the naïve, the implementation of the algorithm presented in [60], the FALLS algorithm, the ScaLAPACK redistribution computation algorithm, and the ASPEN version. The correctness of computed intersection was checked by comparing with the naïve approach results, and on later work by the *Universal Data Junction* library unit tests.

The main loop as shown in Algorithm 1 was timed. In order to limit the impact of system related issues, all memory needed for the creation of intersection description sectors was pre-allocated before any measure of timing was taken. Nevertheless, outliers may appear because of cache misses.

The process grids were made of 2 to 32 processes per grid, and the block-cyclic sizes were one of 1024 by 1024, 256 by 256, 30 by 50 or 654 by 321. The objective was to highlight performance behaviour in regular-to-irregular redistributions, and the impact of partial blocks on the performance. The block sizes were chosen arbitrarily, to range from 1 MiB to 1.5 kB, to vary the number of blocks to be expected and, in definitive, the amount of computation required. The last block size was chosen in order to test the impact of *irregular* sizes, which hardens the computation of common factors. As shown in Figure 3.6, the increase complexity ranges from 6× (ASPEN algorithm, 2×4 to 1×2) to 64× (Guo/Nakata algorithm, 4×8 to 2×16) when changing the consumer block size from 256×256 to 30×50.

ASPEN proved to be very robust over disturbance induced by irregularity in structures. The main influence over the execution time is the number of remote processes per rank. As shown in Figure 3.6, when the number of blocks is scaled by a factor ≈8.5 and ≈5 in each dimension, timings scaled linearly for ASPEN, which is not dependent on the remote number of blocks nor on remote grid dimension, while all other algorithms scale with the square (or worse) of the grid length.

In the case of shrinking the grid by a factor 2 (left cases in Figure 3.6), the second best case is at least 66 % slower than ASPEN (top). The industry standard algorithm (ScaLAPACK) and the FALLS algorithm are generally very close in performance, and show better robustness to high number of consumer blocks, contrary to the naïve approach and to the Guo/Nakata one. The latter result was actually a surprise given the proximity in the

algorithm between Guo/Nakata and FALLS. When combining a complex change in the dimensions of the grid (shrinking one dimension, expanding the second) with the change in block size, the ASPEN algorithm shows a remarkable improvement in performance, being 12 times faster than the next fastest algorithm. The naïve approach, is generally about 14 times slower than the second fastest algorithm in every cases.

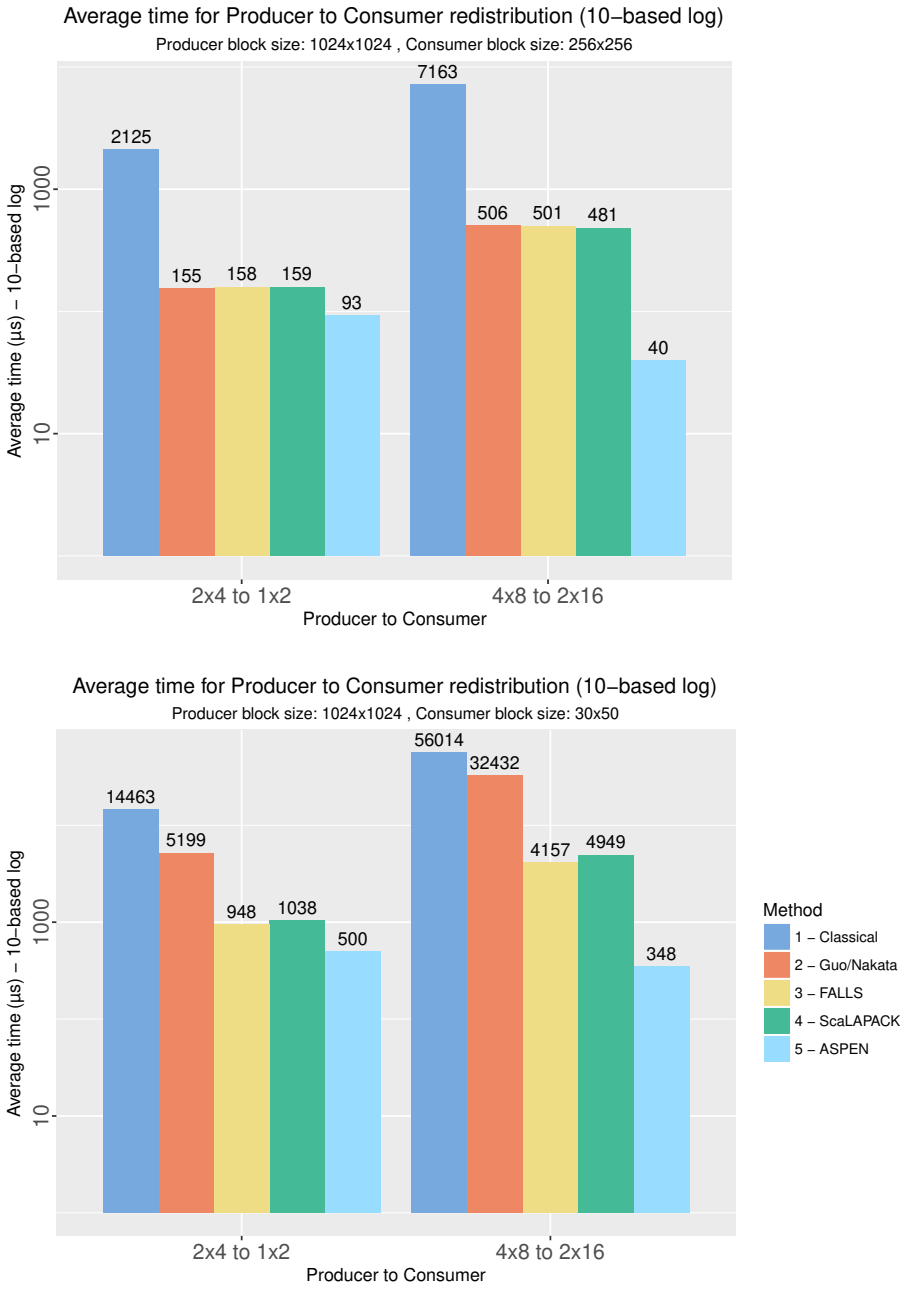
The results shown in Figure 3.7 show the influence of the variation of one dimension of the consumer grid to the performance, and thus of the number of remote processes. These suggest a strong influence on performance from the number of remote processes. Since the R term can become significant even with small grids, we see the performance begin to rise even for modest grid exchanges. With ASPEN, the R term is absent and this effect is limited, hence we can observe a limited variation in the results when adding more remote processes. With large grid sizes, it is expected to see this effect becoming critically significant.

The bottom figure shows the distribution of timings, and therefore, the variability in the measurements. The diamonds correspond to the average value (shown in the top figure) and the bold horizontal line in the box is the median value. Although the timings are fairly short (hundreds of microseconds), the times are not spread too widely around the average. Moreover, the non-overlapping boxes show that our results are statistically significant.

3.6 Conclusion

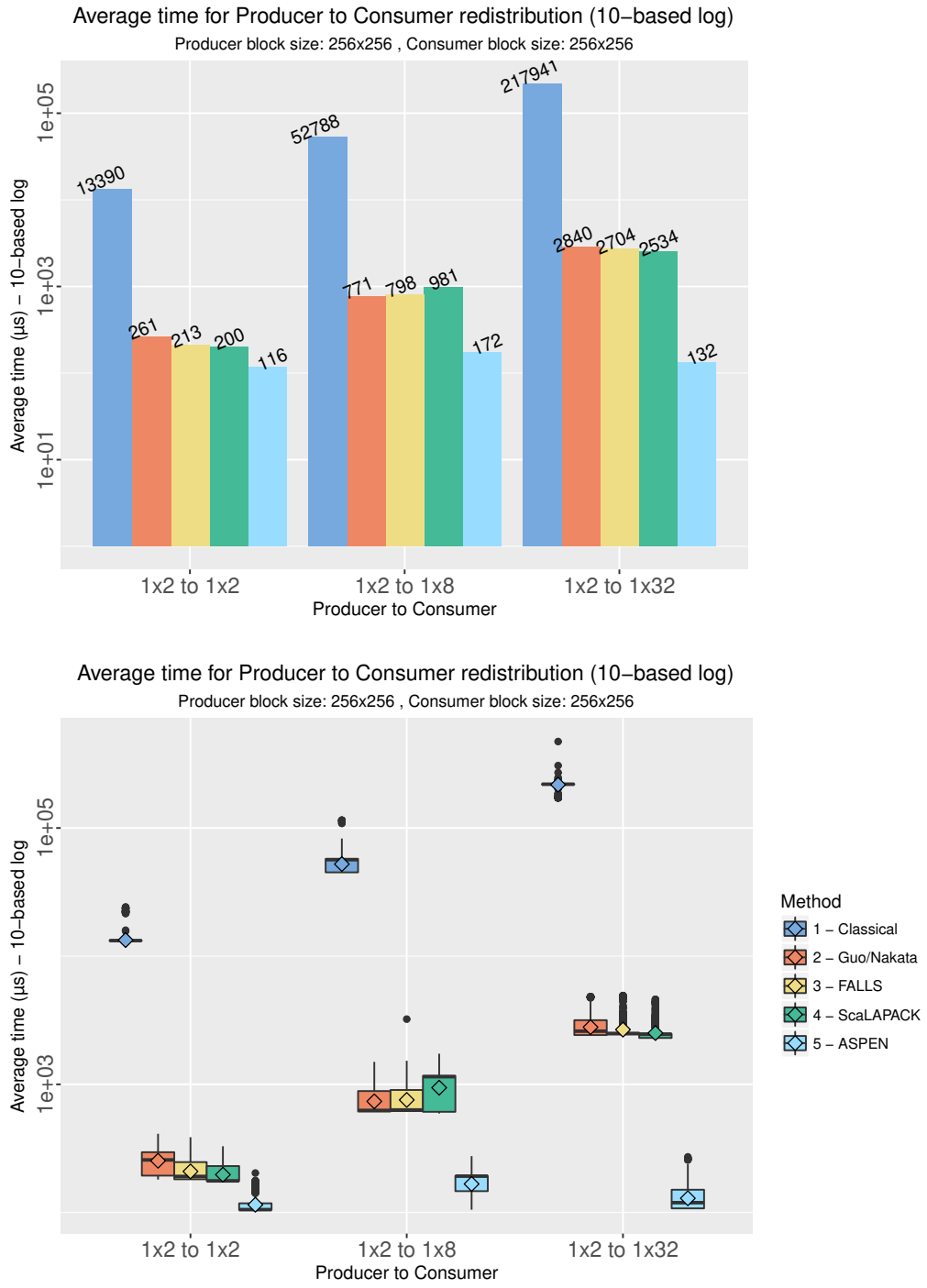
This chapter demonstrated that the ASPEN algorithm can generate redistributions more efficiently (both theoretically and in practice) when moving cyclic data across distinct processor grids. As there is a growing requirement to perform such redistributions across larger grids, the ASPEN algorithm may be impactful. However, the computation time being relatively small when compared to the actual data exchange, the effect may be limited.

As presented in Equation 3.4, when compared to Equation 3.1 and Equation 3.2, a strong theoretical improvement is expected in the indexing phase



Lower is better.

Figure 3.6: Data redistributions for different block sizes and different grid sizes (Producer to Consumer).



Lower is better.

Figure 3.7: Data redistributions for different block sizes and different grid sizes (Producer to Consumer).

of the redistribution. Our small-scale experiments are effectively showing an improvement, and tests from Hsu et al. in [65], which method described in their paper has a complexity similar to ours, are also confirming such results at a larger scale. Moreover, if a currently used algorithm, such as the one used in ScaLAPACK, is expected to scale efficiently for large scale applications, our approach, which does not depend on the number of remote processors, should provide an even better scaling factor.

A direct application of it is used in the UDJ (Universal Data Junction) project developed internally at Cray (now Hewlett Packard Enterprise (HPE)). The total cost of moving data across jobs will depend on many factors, such as the cost of generating the redistribution, the cost of buffering data and message latencies. The UDJ library is used to send complex distributed data across production HPC jobs, and the ASPEN algorithm is used to compute the dimension and offset of data chunks for inter-process communications.

In addition to participating in improving communication, this algorithm can also be used in order to optimise memory management. The heterogeneity of memory technologies and the limited space available makes the matching between data and memory spaces increasingly relevant on the path to exascale.

Yet, managing heterogeneous memory is tedious and usually non-portable as it requires adapting the code and identifying the appropriate NUMA node that corresponds to the targeted memory. In the next chapter we shall present our approach for how to provide a unified memory interface for heterogeneous systems.

Chapter 4

An Abstract Approach for Memory Management in Heterogeneous Memory Systems

Contents

4.1 Introduction

Computing systems have consistently increased in complexity over time. The memory has been diversified and its management has been complexified by the processing technologies, which may embed memory in its package isolating it from the rest of the system. Not all memory can be directly accessed. Exchanging data with GPUs requires the usage of dedicated APIs (CUDA, HIP, OpenCL). In addition, new technologies such as non-volatile memory also require specific support in order to enforce the coherency in addition to the persistence.

As a result, with each new generation of supercomputers, application developers have to spend a notable amount of time porting applications in order to benefit from the improved technologies. Multiple approaches have been studied to leverage this burden: as a framework (e.g. OpenCL, SYCL, etc.), as a language extension (e.g. OpenMP with *targets*) or as a library (e.g. Umpire, SICM, etc.).

Each of these approaches presents its benefits and drawbacks. Using a framework or a language extension may provide good performance, although at the cost of being supported by fewer compilers. Using a framework is a more global approach which requires more work redesigning the code. The library approach provides a great compiler support with often few modifications to the code.

Different needs are addressable in concert. First, the portability issue. A satisfactory solution would be portable across compilers and, if possible, across languages. For this reason, C seems like a good choice as it also provides a good interoperability with C++ and Fortran. The set of functions provided to interact with memory systems is kept minimal to facilitate the integration of present and future libraries. Secondly, a high-level view of the memory system and of the data organisation seems convenient in order to find opportunities to improve performance.

The top-level library, Mamba, has been designed in collaboration with Adrian Tate and Tim Dykes to provide a flexible solution to data layout across heterogeneous memory environment. Its approach to memory het-

erogeneity is to consider it as a graph instead of a hierarchical tree. The execution environment (framework used) is taken into account along with the target location for the data (either CPU or GPU). And for simplicity of usage, automatic detection of the environment is provided, based on the broadly used library `hwloc` [21].

The primary objective of the library is to provide an extensible, modular solution that allows for composition between multiple memory providers. The second objective was to provide a generic framework to support the application of different allocation strategies, effectively extending the usage of pre-existing libraries. The unity of the API also allows for parameter exploration when looking for a tailored solution. It facilitates the comparison between different memory tiers and execution environment, with little change of the code. It may help to choose the right memory provider, the right execution space (CPU or GPU), and the right data layout. One final usage is the possibility to partition and control the memory usage, enforcing at run-time the total allocation of memory, on a per-execution context and per-memory system granularity.

In this chapter, Section 4.2 presents the related work and explains the need for a new solution. Section 4.3 introduces Mamba and the structures deemed necessary to describe the memory and the interactions with it. Section 4.4 shows the evaluation of the Mamba memory allocation system, and finally, Section 4.5 will conclude and summarise before presenting further work.

4.2 Related Work

The effort to provide solutions for application developers has been carried out by many actors of the high-performance computing community, and recommended by the International Exascale Software Project Roadmap [41]. Some frameworks supporting heterogeneous computing environments also provide some support of data locality (e.g. addressing the locality in OpenMP is presented in [67, 107]). Most frameworks lack the support for describing regular data structure and complex indexing like tiling and would require

the user to first reshape the array before moving the data, in contrast to OpenMP since version 5.1.

The Kokkos [47] C++ programming model includes array-based abstractions with polymorphic layouts which may be used to construct tiling abstractions, along with the concept of memory and execution spaces (analogous to memory space and execution context in Mamba), however the extensive use of C++ language features makes such a model difficult to use in C and Fortran.

During the past 5 years many complementary libraries have been developed in national laboratories as part of the on-going effort to reach the exascale. First, SICM (Simplified Interface to Complex Memory) [80] has been created to support a wide range of memory technologies with a unique API. It also provides tools to benchmark some memory characteristics such as the bandwidth and latency for performance analysis. For its wide support of technologies it was important to interface the Mamba library memory management with it. However, data movement and data layout also have to be taken into account to achieve good performance across applications. In addition, memory location is not sufficient when hybrid codes are being run both on CPUs and GPUs. Umpire [19] provides a support for GPU and exposes a single API for allocation and movement of data across the different memory systems. It also proposes a set of generic algorithmic strategies to apply on top of the allocators to improve performance. This library forms the base on top on which the RAJA library [20] is built. AML [114] focuses on memory layout across complex memory topology. It depends on memory segments and data transformation to create asynchronous optimised data movement in order to increase the performance by improving data locality and memory utilisation.

Several libraries have addressed the memory heterogeneity problem [23, 106, 113, 136]. Nonetheless, providing a simplified memory abstraction may be advantageous for an easier integration to applications. As explained in [106], it is considered that the memory broker needs to have a global view of the complete memory system to provide shared data capabilities. However none consider the requirement of knowing the execution target

required to interact with dedicated framework like CUDA.

As the range of compilers supporting the most recent additions to the standards for frameworks and programming models is necessarily restricted, these two approaches were discarded. It was thus decided to provide a library in order to support the different requirements outlined in Section 4.1. Table 2.5 summarises the support provided by the main current programming models, frameworks and libraries.

4.3 Mamba memory manager

A *Mamba Array* (`mmbArray`) is a data container that forms the core abstraction of the Mamba library, introduced in [45]. This array-like object encapsulates user data, which may be allocated internally or can be provided by the user on construction, and is the typical entry point for data allocation, movement, and access via the Mamba library. The structure of a Mamba Array is explicitly described by the user, which enables a flexible mapping of logical array indices to physical byte offsets in a memory space. This allows for coarse-grained data movement on a complex computing system.

The underlying data contained in a Mamba Array may be subset, duplicated, or moved between memories, either explicitly by the user or implicitly by the Mamba runtime. Such a subset is contained in a *Mamba Tile* (`mmbArrayTile`). A Mamba array consists internally of a series of such tiles describing blocks of data (which may or may not be contiguous) stored in a specific memory space and available for access in a specific execution context, illustrated by Figure 4.1. Each tile is independent and can be moved or duplicated independently from the rest of the array.

To support such a structure, a memory management library was developed to provide a portable abstraction of the memory system in order to facilitate the underlying memory operations without the burden of dealing with each library-specific APIs. The library supports C, C++ and Fortran, and provides tools for an easy access of a tiled array, both on CPUs and GPUs.

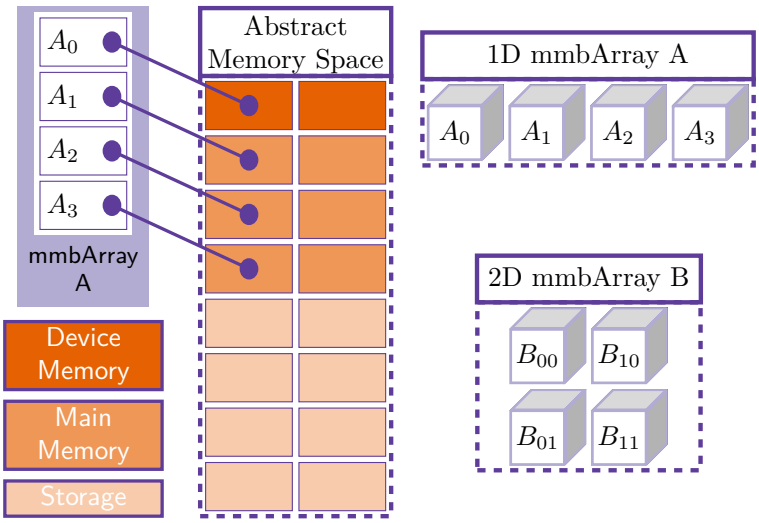


Figure 4.1: Illustrating the concept of tiled Mamba Arrays, where each tile may reside in a different memory space as defined by the memory system model outlined in Section 4.3.1.

4.3.1 Memory System Model

Mamba container objects are based on an abstract model of the memory system that aims to encapsulate the variety of memory types available on, or near, a typical compute node. These are conceptually grouped together into an *abstract memory space* (see Figure 4.2), and different memory types are exposed as memory spaces from which allocations can be made. An underlying generic memory management library implements this memory system model, providing mechanisms to allocate and transport data between memory tiers through a uniform interface. As illustrated in Figure 4.3, each type of memory is considered a memory layer in which memory spaces of a specific capacity may be created. Each space is accessed by a space-specific memory interface, and data allocations are provided within an execution context that describes how that allocation may be accessed. Each of the components is described in more detail in Section 4.3.2.

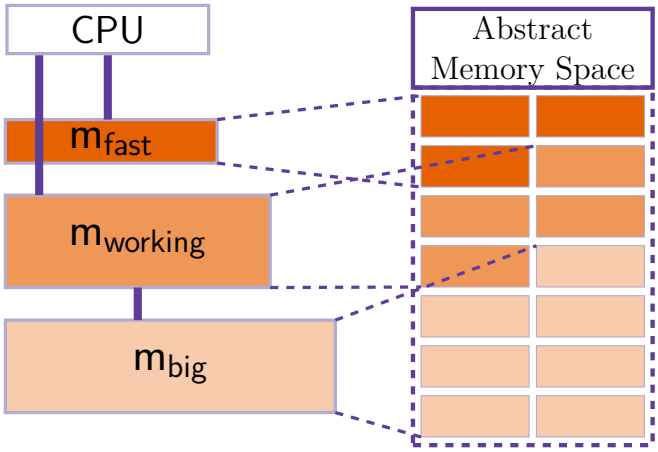
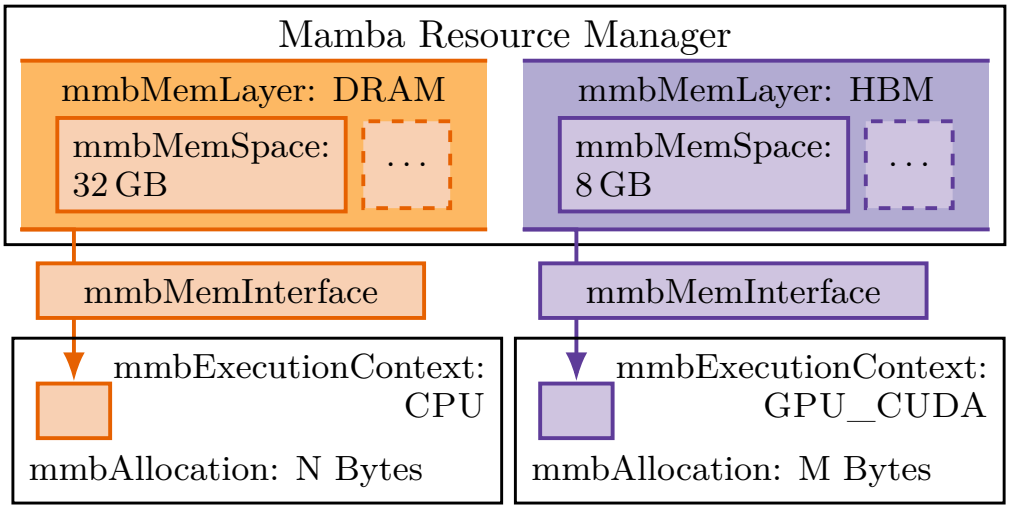


Figure 4.2: An abstract memory space, consisting of three types of memory: $\mathsf{m}_{\mathsf{fast}}$, $\mathsf{m}_{\mathsf{working}}$, and $\mathsf{m}_{\mathsf{big}}$.



An example where two memory spaces exist, each associated to a different memory layer. Each space has an associated memory interface, from which an allocation object may be obtained and used in the appropriate execution context.

Figure 4.3: The data structures that form the memory abstraction in the MAMBA library.

4.3.2 Memory system architecture

4.3.2.1 Layer

A *Memory Layer* (`mmbMemLayer` in the MAMBA API), represents a particular *type* of memory with a defined set of characteristics. Hardware availability defines which memory layers are available and their characteristics such as high bandwidth or low latency.

Each layer's availability is discovered automatically where possible during library initialisation via the `hwloc` library [21]. During the library initialisation, the hardware topology exposed by `hwloc` is parsed, looking for addressable memory systems and devices (GPUs, DIMMs, NVDIMMs). Nodes found which expose the same characteristics are aggregated in order to keep a global view of the system memory, although at the cost of accurateness. However, the detection alone does not ensure the availability of a layer. The availability of a requested layer depends on the support of the memory provider (See Section 4.3.2.2), or on the external library that were given at configure-time.

When the automatic discovery is disabled, the user has to define the spaces (see Section 4.3.2.6) manually using the proper layer.

4.3.2.2 Provider

A *Memory Provider* (`mmbProvider`) abstracts the library (or set of libraries) used to realise the defined micro-set of memory operations. A minimal set of operations has been defined to ensure portability across libraries. The three operations are *allocation*, *deallocation* and *copy*. The library provides a basic support for a limited number of layers (DRAM and GPUs with CUDA support). However, the library provides compatibility layers for other memory libraries such as SICM [80] or Umpire [19]. The MAMBA Memory Management compatibility mechanism enables other external memory providers to be interfaced in order to widen the range of supported hardware while keeping a unique API. A provider is expected to support one or more memory layers, such as the addition of one provider is enriching

the overall support of the library, by supporting a new technology or by adding a new feature.

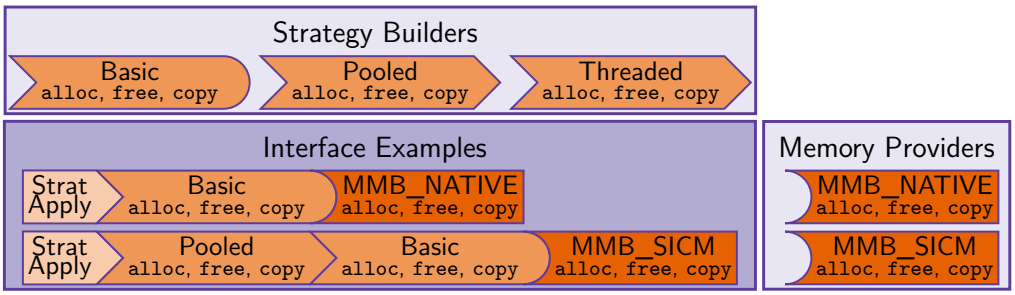
4.3.2.3 Strategy

A *Strategy* (`mmbStrategy`) is a set of algorithms that can be used in order to mitigate the cost of memory calls. It can be used to implement fixed-sized quick allocators, or slab allocators on top of any memory providers. In addition, the strategies can be used to add specific behaviour such as thread safety or utilities such as internal monitoring of memory operations.

The strategies are defined as modules that can be composed in any order. Adding a new strategy requires the creation of one function that enqueues the strategy modules and the respective arguments in the right order. Thus, for example, creating a thread-safe slab-allocator requires enqueuing the thread-related module and a pointer to its arguments, followed by the slab-allocator module and a pointer to its arguments. The execution path is built statically, using C variadic functions. This enables the execution path to be known at compile-time, while providing a modular, flexible and mixable set of independent functions. Hence, the addition of a new strategy can be the result of a new composition of modules, or the addition of a new submodule. One special strategy module (Basic) is responsible for calling the requested memory provider via a static indirection table, and one specific module initialises the variadic arguments dequeueing.

4.3.2.4 Interface

A *Memory Interface* (`mmbMemInterface`) acts as an interface to a specific memory space, providing a unified mechanism to allocate, copy, and free memory. A memory interface may have a specific *strategy*, that defines the behaviour of the interface. For example, this may enforce thread safety during allocation of a critical resource, or define the type of memory allocator used (e.g. pooled vs. slab). The interface also depends on a memory *provider* for memory allocation and deallocation operations. The modularity between memory providers aims to provide the best of the two —



The selection of a strategy enqueues the corresponding strategy modules. The Basic module is responsible for interfacing the strategy with the memory provider requested. The top interface is configured with `MMB_STRATEGY_NONE` as strategy with `MMB_NATIVE` as provider. The bottom interface is configured with `MMB_STRATEGY_POOLED` as strategy with `MMB_SICM` as provider.

Figure 4.4: Schematic view of the `mmbMemInterface` structure.

or more — worlds, by allowing the user either to swap between providers depending on use-cases or to benefit from a special feature. For example, SICM supports memory allocation to Intel Optane™ DC PMM but Umpire offers memory operation logging and replay. Swapping from one provider to the other is as simple as changing the initialisation parameters.

4.3.2.5 Execution Context

With the development of the heterogeneity of computing systems, some memory systems are not available to the processor and rely on their own software stack with asynchronous callback to perform memory operations. A program usually relies on a constructor's API (or the open source equivalent) to manage memory. The *execution context* (`mmbExecutionContext`) is defined in order to determine how memory is allocated and made available in a space. The execution context provides a means of choosing the programming model through which memory must be made available. For example, NVIDIA GPU device memory could be provided within a CUDA, HIP, or OpenACC execution context.

4.3.2.6 Space

A *Memory Space* (`mmbMemSpace`), represents a size-limited, addressable, instantiation of a memory space corresponding to a specific memory layer. The size may be limited by a call to the API or based on the hardware-defined value. Users will typically obtain a memory space by specifying a layer, an execution context and a size, although the memory space may be configured in diverse ways. The space's responsibility is to limit the amount of allocated memory to a fixed value. This enables a user-side software-managed memory-sharing solution. The space is also responsible for keeping tracks of the memory interfaces, acting as a factory when an unavailable strategy+provider couple is being requested.

Memory space configuration The memory space configuration API can be used to specify two aspects of a memory space behaviour. First, the total amount of memory this memory space accounts for. The value can be modified during the execution, although the value cannot be reduced below the amount of memory already allocated. A flag can be used to set the limit to *infinity*. The second aspect of a memory space that can be configured is the default interface to be provided on request. These parameters correspond to the interface configuration options as presented in Section 4.3.2.4. They define the default interface that gets generated when the memory space gets instantiated. Additional interfaces are created on user request with customised allocation strategies (for example, a pooled and/or thread-safe allocator) or using a different provider.

4.3.2.7 Memory Manager

The memory management exposes a centralised structure (`mmbManager`) that may be used as a single entry point to the memory operations via its spaces and interfaces, although the library supports the definition of multiple independent managers. The Mamba library memory system supports both automatically-generated and user-provided descriptions of the memory system. When built with `hwloc` support, the available layers (HBM,

```
mmbManager *mngr;
mmb_manager_create(&mngr);
// Set the space configuration
mmbMemSpaceConfig *gpu_space_config;
mmb_memspace_config_create_init_default(&gpu_space_config);
mmb_memspace_config_interface(gpu_space_config, MMB_NATIVE,
    MMB_STRATEGY_NONE);
mmb_memspace_config_set_size(gpu_space_config, 2UL << 30);
// Request memory spaces for GPU
mmbMemSpace *gpu_space;
mmb_manager_register_memory(mngr, MMB_GDRAM, MMB_GPU_CUDA,
    gpu_space_config, &gpu_space);
```

Listing 4.1: Example of a manual registration of a `mmbMemSpace` for a CUDA based GPU with 2 GiB of memory.

NVDIMM, GPUs, etc.) are identified during `mmbManager` initialisation and corresponding memory spaces and execution contexts are instantiated using the system exposed characteristics. A default memory interface is created for each memory space, however additional interfaces may also be created on user request with customised allocation strategies (for example, a pooled and/or thread-safe allocator). An API is also provided for the user to describe the different memory spaces they expect to be using, in case the library is built without `hwloc` support. Each space requires an appropriate memory layer, a size, and an execution context for construction.

4.3.2.8 Allocation

The *Allocation Object* (`mmbAllocation`) is the result of a successful memory request. It provides an abstract container for a memory allocation in a specific memory space. Allocation objects are provided by memory interfaces, and passed into generic allocation, copy, and free routines. This abstraction is similar to the smart-pointers in C++, and contains metadata about the memory allocation (e.g. size, *ownership* of the underlying data) along with provenance information such as the interface through which it was allocated. The concept of ownership allows, for example, sub-allocations to be created as slices of existing allocations, or to execute operation with the

```
mmbMemSpaceConfig *dram_space_config;
mmb_memspace_config_create_default(&dram_space_config);
// Request memory spaces for cpu and gpu
mmbMemSpace *dram_space, *gpu_space;
mmb_request_space(MMB_DRAM, MMB_EXECUTION_CONTEXT_DEFAULT,
    dram_space_config, &dram_space);
// NULL equivalent to default options
mmb_request_space(MMB_GDRAM, MMB_GPU_CUDA, NULL, &gpu_space);
// Request memory interface
mmbMemInterfaceConfig dram_interface_config =
 {.provider = MMB_PROVIDER_DEFAULT, .strategy = MMB_POOLED};
mmbMemInterface *dram_interface, *gpu_interface;
mmb_request_interface(dram_space, &dram_interface_config,
    &dram_interface);
mmb_request_interface(gpu_space, NULL, &gpu_interface);
// Allocate buffers on host and gpu and copy between
mmbAllocation *host_buffer, *device_buffer;
mmb_allocate(n_bytes, dram_interface, &host_buffer);
fill_host_buffer(host_buffer);
mmb_allocate(n_bytes, gpu_interface, &gpu_buffer):
mmb_copy(device_buffer, host_buffer);
```

Listing 4.2: Example construction of CPU and GPU based memory interfaces, allocating a buffer for each memory space and copying between with unified interface. Memory spaces are assumed to be constructed automatically during library initialisation.

library on buffers not allocated with Mamba.

However, because the C language lacks the polymorphism and overloading capabilities C++ provides, the user has access to the allocated buffer only through either the appropriate field of the `struct` (`.ptr`), or by considering the allocation handle as a pointer to a pointer to the buffer and can directly dereference twice the allocation object to access the data. The handles are allocated independently from the rest of the memory management, as they can be used to wrap user provided pointers or subsections of previous allocation. The allocation is made via a thread-safe slab-allocator that is shared by the different memory managers instantiated.

4.3.3 Memory System Configuration

The library has been developed with the objective of portability, modularity and adaptability. At configuration time, the library tries to find all the supported third-party libraries available, prioritising the path given as a parameter if any. This defines which providers and execution contexts have to be included in the library. Configure time and compile-time also allow for default behaviours to be defined, but these can be overwritten at execution time by setting the environment or during run-time through the API.

The default behaviour can be used to define a preferred GPU framework as execution context, but also the provider and strategy for the default interface created by a space when instantiated. The build-time and compile-time settings define the system-wide default behaviour for the whole library. Environment settings define the behaviour on a per user basis. Calls to the API enable a customised execution during run-time.

This library was designed and developed as a tool for exploring and mixing parameters for application tuning. This led to an easily customisable and extensible library.

4.3.4 Memory Discovery

As stated before, during manager initialisation, the topology of the machine can be automatically discovered using the `hwloc` toolbox. The topology reported exposes the different addressable memory tiers in addition to the caches. The GPU related memory is exposed by setting the flag `HWLOC_KEEP_NVIDIA_GPU_NUMA_NODES` to `1` as the corresponding NUMA nodes would usually be hidden by default. The NUMA nodes are iterated using `hwloc_get_next_obj_by_type(3)`, looking for a subtype (*MCDRAM*, *DaxDevice* or *GPUMemory* for either HBM, NVDIMM or GPU memory respectively). If no subtype is found, the NUMA node is considered as a DRAM memory. NUMA nodes sharing the same subtype are aggregated into one layer, adding all the memory together. Further work may improve the data locality by creating NUMA-aware provider and layers.

Finally, the execution contexts are also automatically inferred by iterating the devices of type co-processor or GPU, as exposed by `hwloc_get_next_osdev(3)`. The backend found states whether the device is managed by CUDA or by OpenCL (the two execution contexts detected for GPUs).

A memory space is generated for each layer with more than 0 B of memory detected. In the case of GPU memory, a memory space is instantiated for each of the detected execution contexts.

The further integration of `hwloc` ≥ 2.3 is planned by considering the `memattr`s in order to provide relative memory addressing by requesting a *best layer* latency-wise, or a bandwidth optimised layer for example.

4.4 Evaluation and results

The support for heterogeneous memories is summarised in Table 4.1, and displayed for comparison with the other similar memory broking libraries. As aimed for by design, Mamba is able to support to support all targeted languages but also all targeted memory tiers. Some of the memory support may rely on other cited libraries (e.g. memkind and libnuma), but our library also adds allocation mitigating strategies, a unified API and support for C, C++ and Fortran.

The performance of the library was evaluated using a selection of benchmarks from the repository in [83]. This set of benchmarks is initially dedicated at comparing mi-malloc [84] with other memory providing libraries. The selection of benchmarks was based on the API required as the current implementation does not provide support for reallocation or memory aligned allocation as an example, while still keeping a variety of allocation patterns. All of the benchmarks have been fitted with a wrapper for the main function in order to parse the memory configuration parameters without disturbing the main application parameters. This allowed having a single application that could be run with any of the currently supported memory providers. In order to evaluate the overhead of the proposed solution, additional versions were developed to interface different memory providers directly. This step required extra work as the different libraries

		CPUs			GPUs		Mitigation Strategies	Data Layout	Languages		
		DRAM	HBM	NVDIMM	NVIDIA	AMD			C	C++	Fortran
Libraries	AML	✓	✓	✗	✓	✗	✗	✓	✓	✓	✗
	jemalloc	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓
	libnuma	✓	✓	✓	✗	✗	✗	✗	✓	✓	✗
	memkind	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗
	SICM	✓	✓	✓	✗	✗	✓	✗	✓	✓	✗
	tcmalloc	✓	✗	✗	✗	✗	✓	✗	✓	✓	✓
	Umpire	✓	✗	✗	✓	✓	✓	✗	✓	✓	✓
	Mamba	✓	✓**	✓**	✓¶	✓‖	✓	✓	✓	✓	✓

** When the library is configured with memkind or SICM.
¶ When the library is configured with CUDA, HIP or OpenCL. ‖ When the library is configured with ROCm, HIP or OpenCL.

Table 4.1: Updated Table 2.5, memory management libraries and memory support.

where exposing a variety of interfaces, e.g. passing by side-effect a pointer to the allocated buffer instead of returning it, or requiring the size of the allocation when deallocating.

Each of the presented benchmarks have been run 10 times, on a Cray XC50 cluster, on a single node with two 22-cores Intel Xeon CPU E5–2699 v4 running at 2.20 GHz, with 128 GB of DDR4–2400 memory. Libraries and benchmarks were compiled with the Cray compiler version 11.0.1 and ran on Cray Linux Environment release 7.0.UP01.

Mamba was configured to support system provided `malloc`, jemalloc provided `malloc`, SICM and Umpire via the C wrapper provided. As most of the benchmarks used are written in C++, the C API was used to interact with the library. Umpire was configured without any GPU support but with support for NUMA enabled. The binaries with the `-mmb` suffix use

the Mamba library. In the figures, the mode is indicated after the '+' sign and corresponds to whether the memory is provided by the library linked (*system*) or by a provider selected through the Mamba library. Times are provided in minutes and seconds, separated by a colon. Times and memory footprint are reported using `/usr/bin/time`

4.4.1 alloc-test

The alloc-test, by OLogN Technologies AG [1], is a very allocation-intensive benchmark doing millions of allocations in various size classes, all less than 1 kB. The test is scaled such that when an allocator performs almost identically on alloc-test1 as alloc-testN it means that it scales linearly. The application was configured to do up to 10^8 allocations in a single thread, and in each of the 16 threads. The experiments were not scaled on all cores because of the disproportionate execution time which would not let the execution terminate because of timing out. The results are shown in Figure 4.5. Only the single threaded version's results are shown. For the multi-threaded version, `alloc-test`, `alloc-test-jemalloc` and `alloc-test-wrapped-malloc` scale without issue. The Mamba version, however, serialises the update to `mmbMemSpace` in addition to the allocation of a `mmbAllocation` handle, which ultimately serialises the allocation requests. The results are similar, although the times are multiplied by 30 while the number of allocations are multiplied by 16.

In addition to the Mamba, SICM, Umpire and jemalloc interfaced versions, an additional version was provided that allocates data with an extra level of indirection. As presented in Section 4.3.2.8, Mamba returns the pointer to the allocated data wrapped into a structure. Hence, accessing the data requires an extra level of indirection when dereferencing the pointer to the `mmbAllocation` structure. This supplementary benchmark is used to evaluate whether returning a wrapped allocation induces an overhead, and found that, for the test case, the overhead was 24 %.

The relatively bad performance of the SICM library is widely linked to the fact that the SICM library only allocates memory using `mmap(2)`

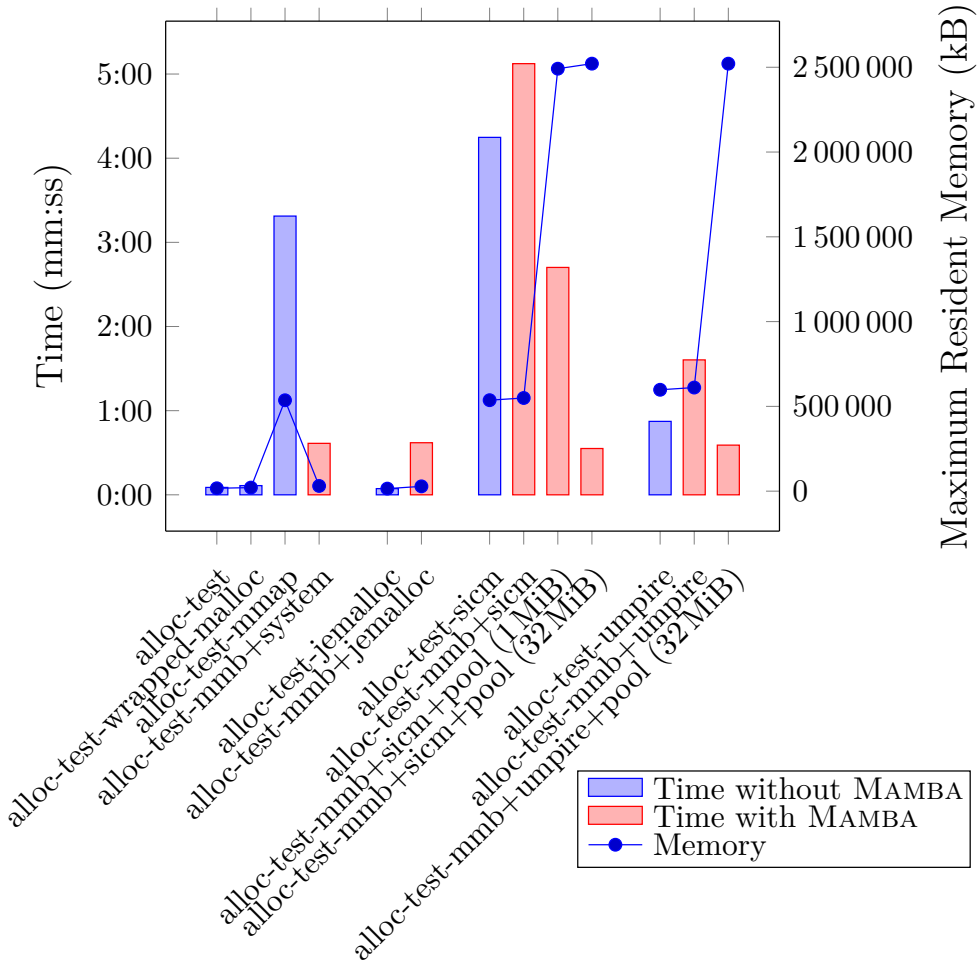


Figure 4.5: Results for `alloc-test` binary, single thread.

despite the small size of the allocated objects (<1 KiB). This analysis is confirmed by the `alloc-test-mmap` benchmark which presents results much more similar to `test-alloc-sicm` than `alloc-test` for example, and a similar memory footprint. The reported timings also show that out of 4 minutes spent by the application, about 3 minutes were spent in system calls.

Although the usage of the Mamba library adds a significant overhead, we observe that the relative difference in performance between the providers is preserved for the heterogeneous memory providers. However, the dif-

ference between the two `malloc` implementations is inverted despite the difference in memory footprint being preserved, along with the difference in page faults. Moreover, the enabling of the pooling strategy for making fewer calls to the provider libraries improves drastically the performance. The reported performance for the SICM implementation shows the best improvements. For a slab size of 1 MiB, the performance of the allocation is improved by 50 %. A 32 MiB slab allocation reduces the allocation overhead to a similar level as using `malloc` via Mamba, disregarding the underlying memory provider. The size of the memory pool is defined at initialisation using the `mmbOptions` parameter.

Even more, our experiments show that using libraries wrapped in Mamba along with strategies can improved the performance compared to using the library alone. For both SICM and Umpire, using pools of 32 MiB via the Mamba library, we achieved better performance than we did with the native library allocation. We can then conclude that Mamba may allow for easy and fast benchmarking for both multiple libraries and multiple strategies, without requiring the tedious work of rewriting the implementation of the allocating function wrapper in all the different cases. The flexibility of our solution provides support for an easy configuration.

The difference in performance is due to the number of memory pools that are being required. For a 1 MiB pool, 2462 were generated, while 80 only were required with a size of 32 MiB. However, the search for a suitable pool is linear in the number of pool for each allocation and all newly allocated pools are added at the end of the queue which adds up a significant overhead to the research process. The strategies that were developed are mainly proofs of concepts, their implementations have room for performance improvements.

4.4.2 malloc-large

This benchmark tests the allocation of a 2 MiB memory block, with its writing, in order to ensure the creation of the memory pages. The original version of this benchmark produced odd results for the version based on

the system provided `malloc`. Following the examples shown in [25], in order to ensure that the loop writing the data in the newly allocated buffers would not be optimised out, empty volatile assembly code was inlined before and after the writing loop. The timing produced in Figure 4.6 is the timing of the whole application execution. One point of interest of this benchmark is that the buffers are embedded into `unique_ptr` C++ containers and automatically discarded, and the memory freed, between iterations. This enabled the evaluation of the complexity of integrating the library into a modern C++ workflow. Moreover, this test-case enabled the evaluation of the base memory footprint of the library at around 10 MB, with no allocation. For the series of 5000 allocations, we observe that the SICM library consistently generates over 2 560 000 minor page faults, without the use of huge pages, both when used natively and when used with Mamba. A minor page fault happens when a page is present in memory but not checked in the memory management unit. This locality issue is probably the origin of the substantial difference compared to any other versions of the benchmark.

4.4.3 larson

The larson server benchmark by Larson and Krishnan [81] allocates and frees between threads. They observed a behaviour they called bleeding in actual server applications, and the benchmark simulates this.

The different tests show an overhead of 3 % when using Mamba compared with the system provided `malloc` allocation. Contrary to other tests, the number of minor page faults seems stable when the memory is allocated with SICM, and no major difference in timings is observed in Figure 4.7. This result shows that for more realistic workloads, the library has a limited negative impact on performance, while enabling more flexibility and heterogeneous support. As with the `alloc-test` case, the `larson-wrap` version uses the system provided `malloc` implementation, but returns a pointer to the buffer pointer. This was used to evaluate the impact of the added indirection on the benchmark performance. This impact is shown to be below 1 %.

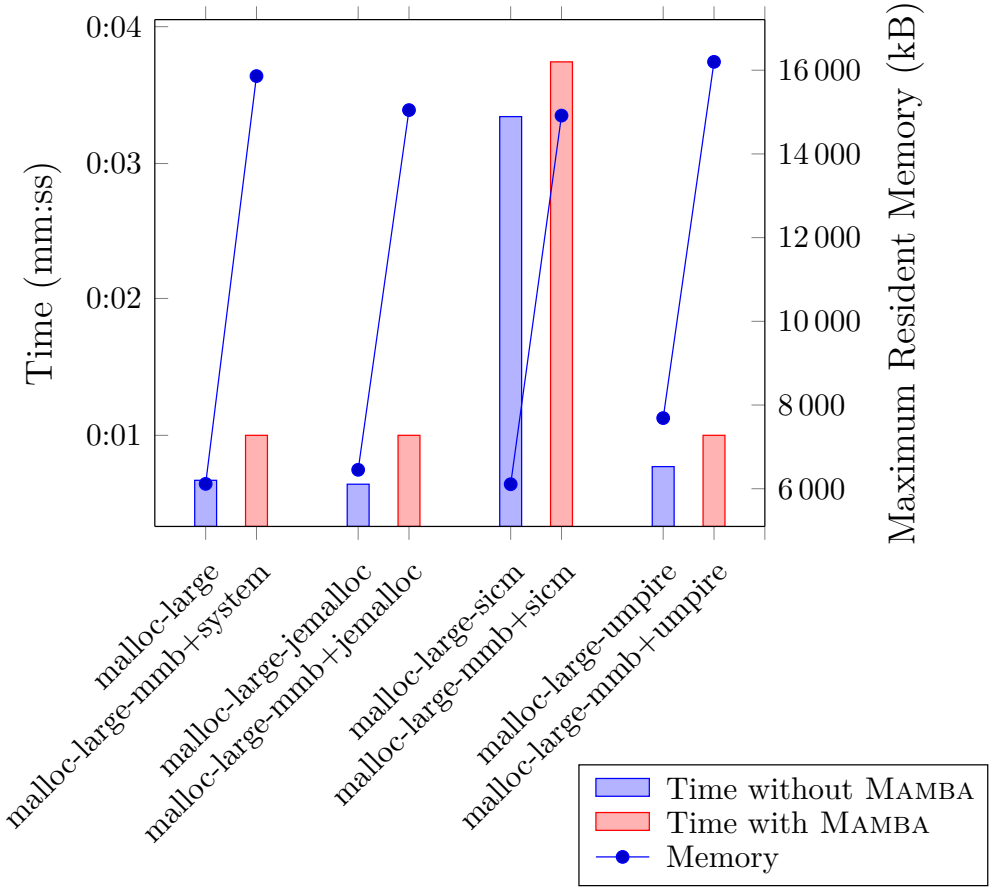


Figure 4.6: Results for the `malloc-large` benchmark, single thread.

4.4.4 xmalloc-test

This test was developed by Lever and Boreham [86] and Christian Eder. An updated version from the SuperMalloc repository [2] is used and it was extended to adapt it to the evaluated set of allocators. This is a more extreme version of the larson benchmark with 22 purely allocating threads, and 22 purely deallocating threads with objects of various sizes migrating between them. This asymmetric producer/consumer pattern is usually difficult to handle for allocators with thread-local caches.

Contrary to other benchmarks, this one is written in C instead of C++. It seems to impact negatively the performance of the Umpire library due to

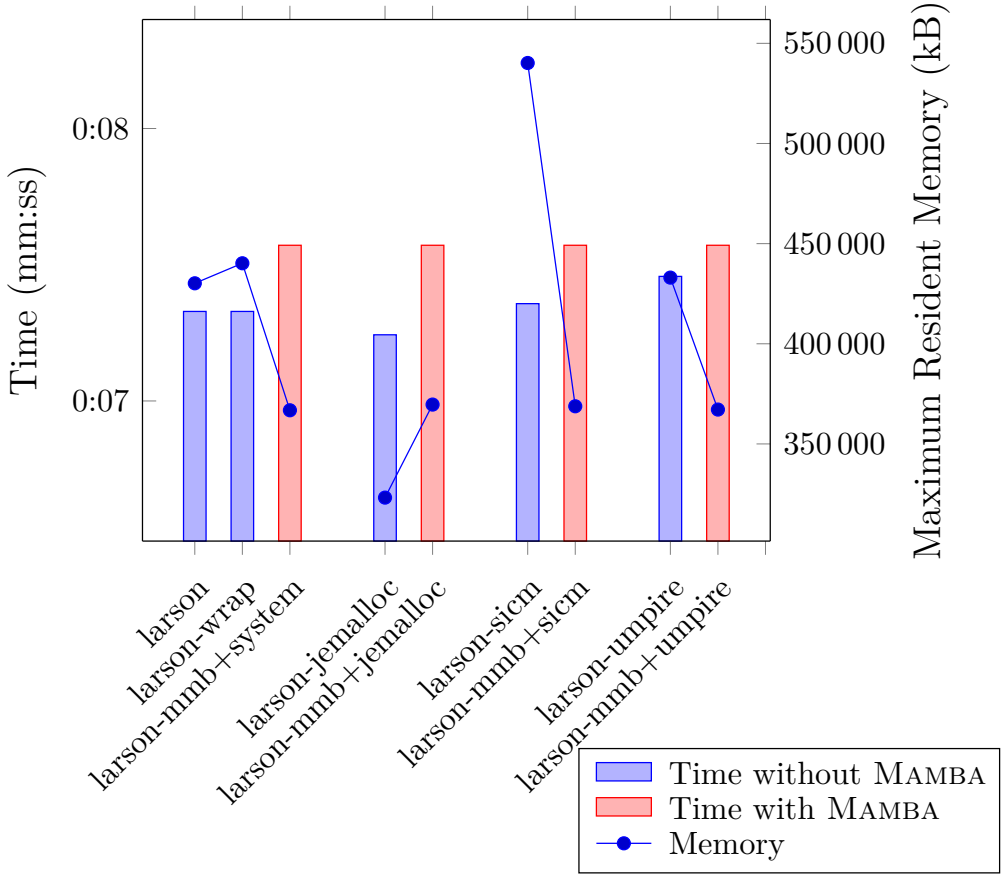


Figure 4.7: Results for the `larson` benchmark, 44 threads.

the wrapper that may disallow some optimisations. Micro benchmarking further, using a C++ intermediate module to provide the allocation functions, a 3.6 % improvement appears in favour of the C++ version, both in time and in the number of `free` operations realised per second. However this result did not show in the overall timing of the execution.

Overall, this benchmark also shows a limited overhead when allocating and deallocating memory using our library as the penalty is about 8.2 % in the worse case (`xmalloc-test` against `xmalloc-test-mmb+system`).

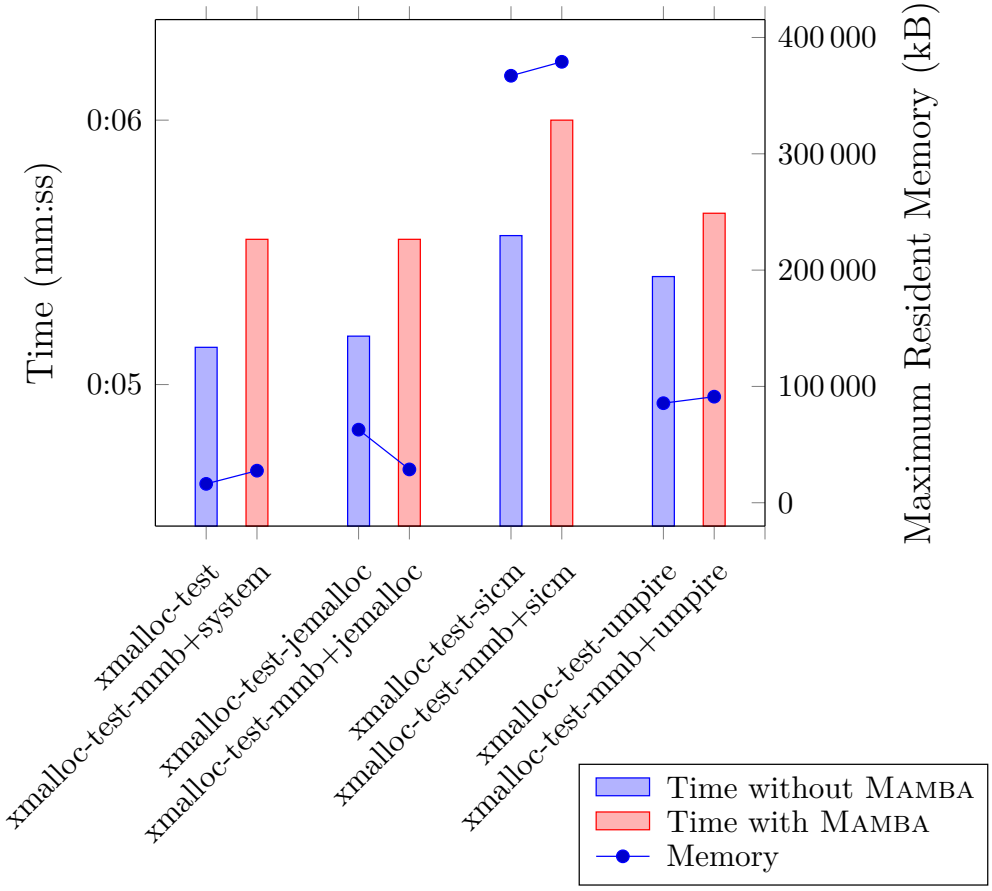


Figure 4.8: Results for the `xmalloc-test` benchmark, 44 threads.

4.5 Conclusion

This chapter presented a tool created with the objective of simplifying the development, maintenance and evolution of applications with respect to the ever more complex memory environment. The metadata describing the layout of each array is carried throughout the program and into the library to optimise data locality. In addition to providing a unique API for allocation, deallocation and copy of data for a range of standard libraries for better library composability, it also provides a great flexibility to simplify parameter exploration in order to tune an application. Signatures differ between memory providers — e.g. the deallocation function may require

the allocated size, or the allocation function may return the value by side-effect — hence writing comparative benchmarks can be a tedious task. Providing a simple and unique memory management interface may facilitate the convergence between the multiple efforts to address the data locality in question in HPC systems [130].

Results show that despite a high overhead for highly concurrent, highly intensive allocations, the usage of basic mitigation strategies can improve overall performance, while providing support and compatibility across libraries. For less intensive workloads, the measured overhead is between 3 % and 8.2 %.

Moreover, this library enables mitigation strategies and algorithms to be implemented once and expected to work with any additional memory provider. This allows for an adaptability of the code to emergent technologies and libraries. Using mitigation strategies can improve overall performance despite the library overhead. Associated to the automated tiling and data layout and placement tools, this library simplifies the prototyping of applications and the tuning for performance.

Future work includes a better use of *memattr*s from *hwloc* in order to provide relative memory selection (e.g. request a copy of data to a better memory location bandwidth-wise). The addition of a support of our library for Python is also being studied. This library is expected to provide improved data locality which would improve the performance of applications on the newest platforms.

In addition to a high-level description of the memory system and its usage, some algorithms may provide a well-known pattern of memory transfers. The next chapter presents the case-study of implementing a memory management system in C for a high-level language such as Python.

Chapter 5

Enabling System Wide Shared Memory for Performance Improvement in PyCOMPSs Applications

Contents

Python has been gaining traction for years in the world of scientific applications. However, the high-level abstraction it provides may not allow the developer to achieve their peak performance. To address this, multiple strategies, sometimes complementary, have been developed to enrich the software ecosystem either by relying on additional libraries dedicated to efficient computation (e.g. NumPy) or by providing a framework to better use HPC scale infrastructures (e.g. PyCOMPSs).

This chapter will present a Python extension based on SharedArray that enables the support of system-provided shared-memory and its integration into the PyCOMPSs programming model as an example of integration to a complex Python environment. The impact such a tool may have on performance is evaluated in two types of distributed execution-flows, one for linear algebra with a blocked matrix multiplication application and the other in the context of data-clustering with a k-means application. The results show that with very little modification of the original decorator (3 lines of code to be modified) of the task-based application the gain in performance can rise above 40 % for tasks relying heavily on data reuse on a distributed environment, especially when loading the data is prominent in the execution time.

5.1 Introduction

Through the convergence between HPC (High-Performance Computing), AI (Artificial Intelligence) and Big data, one area of focus is the availability of common tools that can bring the performance of the former to the techniques and algorithms used by the latter two. One big actor of this evolution is the development of the Python ecosystem, which provides a large set of tools and libraries while being designed for code readability, maintainability and enhancement over time. However, the ease of use comes at the cost of a complex memory management behind the scenes, and complex data structures that get abstracted for the user. The computation intensity required by scientific applications cannot suffer such an overhead to be able

to provide performance. To overcome this, we use a dedicated data structure that provides contiguous buffers to make the best of modern processors and architectures.

In addition, frameworks like COMPSs [89] provide a seamless way to parallelise workloads across large scale infrastructures, notably for Python with the binding provided by PyCOMPSs [125]. However, because of the complexity of Python's internal structures, the communication between nodes and between processes is more complicated compared with using languages such as C or Fortran. Python's class internal structure relies heavily on pointers to numerous structures, which could not be easily shared between processes. In order to benefit from Python memory allocation mitigation strategies, the fine grain management of memory can depend only on the Python interpreter, but care has to be shown when dealing with internal reference counters due to the risk of early deallocation done by the garbage collector.

In this chapter is presented a Python extension based on SharedArray [97] that enables the support of system-provided shared-memory for Python arrays, and its integration into the PyCOMPSs programming model. This demonstrates how the CPython interface in conjunction with the NumPy library can provide tools for memory management outside the Python interpreter, and how to integrate it in a complex framework. Moreover, a way these capabilities can optionally be manually tuned by the user with Python meta-programming features is shown. Related work is presented in Section 5.2, while the design and implementation details will be explained in Section 5.3. Section 5.4 will present the performance evaluation of the solution over the original PyCOMPSs version and Section 5.5 will conclude and present further work.

5.2 Related Work

Multiple approaches for shared-memory have been proposed in the literature. OpenMP [35, 108] supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a

portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer. Since OpenMP is a shared-memory programming model, most variables in OpenMP code are visible to all threads by default. OpenMP was extended to support tasks and their data dependencies. The tasks, as other OpenMP constructs, can operate on private arguments, but also have access to shared variables.

OpenSHMEM [29] is an effort to create a specification for a standardised API aimed at unifying the different SHMEM libraries available. SHMEM is a communications library that adopts the PGAS (Partitioned Global Address Space) programming models. The key features of SHMEM include a one-sided, point-to-point and collective communication, a shared-memory view, and atomic operations that operate on 'symmetric'[1] variables in the program.

However, previous approaches do not support the Python programming language. In Python there are two main issues to overcome when accessing data from parallel tasks: one is the GIL (Global Interpreter Lock), a mutex that protects accesses to Python objects, preventing multiple threads from executing Python bytecodes at once [43]. The second issue is related to the impossibility of accessing Python objects with a reference address. This is only possible with NumPy objects, which can be accessed through a link in a C memory space.

The Python 3.8 multiprocessing module provides the SharedMemory class for the allocation and management of shared-memory to be accessed by one or more processes on a multicore or SMP (Symmetric Multiprocessing) machine[2]. The class permits distinct processes to potentially read and write to a common (or shared) region of volatile memory. The design is similar to the solution used, both for the shared-memory and the integration with NumPy. However, a fine control over the memory management (authorising read-on-write with no synchronisation) and the sharing mech-

[1] Variables are not globally visible, but there are strong assumptions about where they are located across nodes, making them effectively visible.

[2] https://docs.python.org/3/library/multiprocessing.shared_memory.html

anism (System-V or file backed-up with a customisable path) is needed along with portability with Python 2, hence the choice of using a custom approach with a Python extension.

Python 3.8 also provides other means of data sharing between processes and their children. Although all workers are spawned from a common *parent* process, using it to check the type of variables in order to add them to shared-memory would have been a breach of abstraction. It would also have been needlessly complex and inefficient as it would have added an extra access to the serialised version on file to check on the types.

The Plasma In-Memory Object Store[3] is another approach that supports holding immutable objects in shared-memory so that they can be accessed efficiently by many clients across process boundaries. Plasma supports two APIs for creating and accessing objects: a high level API that allows storing and retrieving Python objects and a low level API that allows creating, writing and sealing buffers and operating on the binary data directly. A drawback of Plasma is that it does not support user defined objects, and previous tests with PyCOMPSs did not succeed.

5.3 Design and Implementation

This section will present the design that drove the implementation decisions and the pre-existing framework on top of which the solution was developed.

5.3.1 PyCOMPSs

PyCOMPSs is the Python binding for COMPSs, a task-based parallel programming model for distributed computing platforms. In such a paradigm, the unit for parallelism is the *task*. Tasks are identified by the application programmer, who also indicates the *directionality* of the task parameters and, if required, other metadata such as the type or collection size when applicable. The parameter directions can be *IN*, *OUT* or *INOUT*. Some of

[3] https://arrow.apache.org/docs/python/plasma.html

the metadata can be inferred by the static analysis of the code, but sometimes it is mandatory for the user to make it explicit. With this information, the COMPSs runtime builds at execution time a task graph, where nodes denote task instances and edges data dependencies between tasks. From this task graph, the COMPSs runtime is able to decide which tasks can be executed in parallel and which ones must be executed sequentially. It then performs all the actions required to execute the application tasks on the computing platform. The COMPSs runtime is deployed as a master-worker application, by instantiating a master process on one node and multiple worker processes on others. The master orchestrates the application execution and makes decisions while the application tasks are executed by the worker processes.

The COMPSs runtime is written in Java, but exposes layers of binding for C and Python. The Python specific binding is called PyCOMPSs. As previously stated for COMPSs, PyCOMPSs applications can be executed on distributed computing platforms (i.e. clusters or clouds). Yet, the COMPSs runtime gives the programmer a virtual single memory space. To this end, the COMPSs runtime transfers the data from one node to another when needed. In order to reduce the number of transfers, the locality is exploited as much as possible. However, when the application is being run across multiple nodes, they do not share a common shared-memory space.

While in Java a single process per node with multiple threads is deployed, in the case of Python applications the process of transferring data between tasks is especially delicate. Due to the Global Interpreter Lock, which prevents good parallelisation schemes when using threads as the access to the critical resource serialising the executions, all worker nodes are started as independent Python processes. Hence, the different Python processes do not share the same memory space. As a result, in order to be able to transfer data between two processes, the object to be sent from one task to another is serialised and written into a file. The task that needs the object will read the file and deserialise the object. Additionally, if the processes are in different nodes, the file containing the serialised object needs to be transferred from one node to the other.

Thus, any optimisation to sharing data within one node is critical to the PyCOMPSs worker performance, and providing node-local, system-wide shared-memory support could be greatly beneficial for the application's performance.

The tasks themselves are distributed to the different workers by the Java master. The task details, such as function name, parameter values or filenames, are sent to one specific Python instance on each node that is responsible for distributing the tasks to the members of the Python multi-process environment. This process is also responsible for the synchronisation and termination of the task executing processes. The scheduling decisions depend on the availability of workers and on the data already held by them from previous tasks. However, there is no weighting of the cached data depending on their size. Hence, a single-entry array is weighted the same as a 1 GB array in the decision making process.

5.3.1.1 Objects as files

As the COMPSs model requires communication channels across nodes and across languages and software stacks, it needs a portable way to exchange data. In addition, the synchronisation between tasks is done based on data, which may need to be buffered after being generated and before getting used. To solve this issue, the framework serialises the data into files, and names them following a naming scheme which reflects both the unique data identifier and its *version*. The unique data identifier is unique across the whole application. The *version* of data is a counter that gets incremented every time the data is modified. That way, tasks depending on the new state of the data can still be spawned before all the tasks depending on the previous state of the data have started.

Although providing a lot of flexibility, this management also limits the possibility of using shared-memory based on the data identifier and version unique key. In order not to overwrite data, any memory sharing capabilities would have to maintain this versioning capability, or would need to limit it to read-only variables.

One other constraint to the shared-memory extension is the capacity to release the memory when it is not needed anymore. As scratch space is limited, so is system provided shared memories. If the data is mapped by the system and backed by the file, it may need some costly I/O (Input/Output) operation from memory to disk when physical memory is necessary. If the data is mapped using the SHM POSIX API, the data is pinned and cannot be swapped. In addition, because of the impossibility of applying swap operations, the total amount of shared-memory available is limited (although configurable by the system administrator). Whether for costly operation avoidance or for resource freeing, the need to release memory on demand or when the data is not to be used anymore is essential. COMPSs uses the versioning of data and the task scheduler information to release unnecessary memory and scratch-memory space. PyCOMPSs also exposes an API to explicitly request the deleting of a file or an object, effectively releasing the resources held.

5.3.1.2 Python Decorator

As part of its model, PyCOMPSs uses Python decorators in order to annotate functions and describe some characteristics that cannot be inferred. The **`task`** decorator selects the methods that will become tasks at execution time. Decorators are also used to give hints about the parameters, such as size or datatype, leading to a better parameter management. Describing the directionality of data, the availability or the type of data, or the data structures are examples of metadata to be added using the decorator. I extended the metadata collection so the data can be marked as *recurrent read-only* (see Figure 5.1). The annotated Python code is called *taskified*, as the selected method will be distributed to the different workers at run-time.

The `RRO` flag is only applied to objects which are instances of NumPy class *ndarray* or whose class inherits from NumPy class *ndarray*. Task arguments marked as such will be loaded into the system shared-memory or retrieved from it if previously added. This provides much better performance to the whole system compared with adding every array-based object,

```
1 @task(c=INOUT)
2 def multiply(a,b,c):
3     import numpy
4     c += a*b
```

(a) Task decorator before. The type and directionality for `a` and `b` are automatically inferred.

```
1 @task(a={Type:IN,RRO:True},
2       b={Type:IN,RRO:True},
3       c=INOUT)
4 def multiply(a,b,c):
5     import numpy
6     c += a*b
```

(b) Task decorator with *recurrent read-only* flag enabled. `RRO` can be replaced with the string '`recurrent_read_only`'.

Figure 5.1: Example of usage of the new decorator.

as the loading to the shared-memory adds an overhead to the inevitable deserialisation. Details are provided in Section 5.3.2.1.

It is however possible to mitigate the extra cost with a high reuse of the data. From the original PyCOMPSs interface, each task parameter has to be deserialised when starting a new task, potentially requiring that it be read from file. Applications that rely on multiple reads of the same data through successive iterations (e.g. for data mining applications) can gain in performance when run as tasks using PyCOMPSs as shown in Section 5.4. Moreover, the addition of a simple decorator keeps the high productivity provided by the programming-model [7].

5.3.1.3 Internal Dictionary

In order to keep track of each shared-memory segment, each worker has its own lookup-table to track which data are already mapped in memory. Each worker being its own Python process, there is no need for synchronisation as the memory address space is not shared. Following the Pythonic way, the testing for an existing reference is done by first checking in the dictionary to see whether the entry exists. If not, the algorithm tries to load the requested entry from shared-memory. The shared-memory allows the library to access data that has been deserialised and shared by another process. If the target entry does not exist, an exception is raised by the runtime. This exception triggers the deserialisation of the requested array and the creation of the proper shared-memory segment. There is no

synchronisation across the different workers on one node. However, as the deserialised data are necessarily the same because of the naming scheme, a read-on-write operation would not risk creating an incoherent state. If the array has to be either loaded from memory or deserialised from file, then the corresponding entry is added to the lookup-table. Each entry is indexed by its runtime-defined file name, based on its identifier and its version. The runtime guarantees that this name is unique and kept across tasks if the variable is only read during concurrent or previous tasks. If the variable is modified, the name of the variable is ensured to be different as explained in Section 5.3.1.1. This dictionary is also used in order to deregister all memory segments on request or once the application terminates.

5.3.2 SharedArray, a Python extension

In order to provide the persistence on the working nodes and the sharing ability between processes, an external library was required to interface with the operating system. The library SharedArray provided most of the characteristics one could hope for in this case. This extension to Python provides the interface to create arrays shared either via the POSIX SHM API, or with the memory mapping of a file. The library uses the NumPy [131] library with a CPython interface. However, some limitations made it unsuitable in its current state.

NumPy is a library widely used in order to improve the performance of Python applications. It provides an interface to interact with the data without requiring any copy to memory in addition to using a contiguous buffer. This buffer can be externally provided using the C API of the library. The CPython part gives a native interface between C and Python, and is the entry point to any C-based extension library for Python.

5.3.2.1 Module API extension

In order to give access to the system's shared-memory to Python applications, an extension to the base language was necessary. While [97] is quite thorough, it only allows the creation of zero-initialised arrays. I de-

cided to extend this library in order to provide an enriched API that adds a copy constructor to NumPy based arrays.[4] The CPython API provides access to the internal state of the variables, objects and arrays inside the Python interpreter, and allows the SharedArray library to alter them. This enables the creation of a new object of class *ndarray* with its contiguous buffer pointer referencing the newly allocated shared-memory region instead of the original buffer. The shared buffer is initialised with a copy of the values contained in the original array whose internal reference counter is decremented. However, this implies that the original array used for the deserialisation of the file must still be freed along with its corresponding internal data structures.

In addition to the copy constructor modification, the behaviour for pre-existing names had to be changed as well. Previously, an exception was raised when a name was already taken while registering an array. Preventing this exception to be raised would have required an external synchronisation on the Python side and the addition of a global lock at the node level to ensure mutual exclusion when accessing and loading objects to memory. Moreover, as the shared-memory can be based on the mapping of a file, which could belong to a parallel file system, the synchronisation would have needed to be done across all Python workers, potentially across multiple nodes. This would have added unnecessary complexity to the framework and extra communications costs. As presented in Section 5.3.1.3, and because the read-on-write risk was limited, the race condition on deserialisation was ignored. The same file being deserialised, data coherency is ensured although the location is shared, which prevents an unnecessary synchronisation.

5.3.2.2 Shared-Memory Model

There are many standard ways of using the system's shared-memory. Two options were primarily evaluated. The first method uses `mmap(2)` to create a file-backed up shared segment. The processes mirror the file, mapping

[4]The extended version of the SharedArray is publicly available at https://gitlab.com/cerl/third-party-contributions/shared-array.

it into their virtual memory space. Any modification to the memory is eventually propagated to the file to maintain coherency between processes. The second method uses `shm_open(2)` to create a memory segment held by the system that is persistent after the program ends if not freed. The new memory page created by the system can then be mapped into multiple processes' memory space.

Although the `shm_open(2)` shared-memory implementation exposes limitations unlike `mmap(2)`, the higher performance allowed by the absence of disk operations encouraged the choosing of the former. It also has the advantage of resolving issues due to potential parallel file system name conflicts across cooperating nodes and high latency induced by the environment. Moreover, using the `shm_open(2)` shared-memory API imposes restrictions on the maximum number of segments that can be kept at one time, and the maximum size of one segment. The values can be usually found in the `/etc/sysctl.conf` system file, and defaults to 2 097 152 pages of 4096 bytes which limits it to 8 gigabytes, across 4096 segments of shared-memory. But these limitations, either imposed by the operating system or by the environment, can be overcome by having the settings changed by the system administrators.

5.4 Results

The tests were executed in the MareNostrum III cluster [26], hosted at the Barcelona Supercomputing Center. The codes ran on two nodes. One is the master node, run in exclusive mode in order to avoid sharing resources with any worker, while the second node executes the tasks with 16 processes (workers), one process per core. Each node features two Intel SandyBridge–EP 20M E5–2670/1600 8-cores at 2.6 GHz. As the design relies on the system shared-memory, the decision was made to restrict the execution to one worker with 16 processes on a single node as all cores would be used without over-subscription and would best show the limitation of the system if attainable. The objective of the experiments is to show an improvement in execution time by reducing the overall time required for deserialisation. In

order to limit the influence of external parameters that would add noise to the time measured, all data were stored locally on the nodes' disks, without using the parallel file system.

The first application to be tested was k-means, as the memory access is simple and quickly shows the improvement that can be achieved with careful selection of data for reuse. The second test-case is a task-based blocked 2D matrix multiplication, as it presents some advantages of reusing data but with a more complicated data access pattern.

5.4.1 K-means clustering application

K-means is a widely used clustering algorithm often used by machine learning applications. This algorithm is a numerical, unsupervised, non-deterministic, iterative method that partitions a set of points into k clusters, centred on one point. Each point belongs to exactly one cluster and contributes to the *centre*'s position.

K-means applications have many parameters that influence the behaviour of the application. For the purpose of testing, only four variables were evaluated. These variables were chosen as they were expected to present the most significance in showing the effect of this contribution. The application was run with a fixed number of *fragments*, corresponding to the number of processes acting as workers. Each *fragment* represents a subset of the full set of input points. It is an arbitrary selection of $\frac{\text{number of points}}{\text{number of } \textit{fragments}}$ points to distribute evenly the work-load between tasks. Further work could involve testing the influence of increasing the number of *fragments* without modifying the number of workers and the effect of loading from the system shared-memory compared to loading from the file-system. However, this test requires proper management of the task scheduling as it would require the task to be loaded to a new worker in order to expose the need for deserialising the corresponding subset of points before its addition to the internal dictionary.

The details of the different test-cases and their timings, both with and without the shared-memory extension, are gathered in Table 5.1 on

page 118. The test separately studied the variation of the number of points, of the maximum number of iterations, of the number of dimensions and of the number of *centres*. The result tables show the average timings, but also the 95 % confidence interval of the difference in means, along with the p-value, calculated with the Welch's t-test.

The default number of *centres* is set to 4. This parameter influences the amount of computation and the time spent on each task, as each point has to be compared with each *centre* in order to define its cluster. Increasing the number of *centres* increases the time required per iteration but has very little influence on the time required for serialising and deserialising the data. The *centres*, however, also need serialisation and deserialisation; their number is usually negligible compared to the number of points per *fragment*. In addition, since the *centres* are modified between two iterations, the framework would not allow any gain from reuse of memory and the array would always need to be deserialised. However, diminishing the number of *centres* increases the risk of early termination too much because of the convergence criterion. The dummy test-case 1 from Table 5.1 that only requests one *centre* shows this effect, as the convergence happens in two iterations, as shown in the traces gathered (Figure 5.2-a). The present study does not show much influence of the number of *centres* in percentage of improvement (cases 1 to 5) as the number of *centres* is lower by several orders of magnitude compared to the number of points. The performance improvement is expected to degrade for a smaller $\frac{\text{number of } \textit{point}}{\text{number of } \textit{centres}}$ ratio as the fraction of total time dedicated to I/O will decrease as well.

The maximum number of iterations influences the overall time of the application. For a data-loading/unloading dominated application such as k-means with PyCOMPSs, this parameter would artificially increase the performance gain. In the studied cases, the increase of the maximum number of iterations (case 14) only increases the application running time, with little difference in performance compared to the reference (case 4). The difference in means changed from 9–10 % for case 4 to 10–11 % for case 14.

Finally, the last two parameters are the number of points and the number of dimensions. These parameters have a direct influence on the per-

formance gain as they both define the amount of data to be loaded, and thus, the potential improvement by keeping this data in memory instead of loading and unloading it at each iteration. *point* and *centre* are vectors of doubles. Each vector's length equals the number of dimensions. The number of dimensions usually represents different variables influencing the points being clustered. The number of dimensions only influences the amount of data in the *fragment* and *centres*. However, the number of points also influences the amount of data to be loaded when returning the *labels* array which contains the affiliation of each point to one of the *centres*.

The points used for the tests were generated randomly, with a constant seed shared across cases. The points were generated using a uniform random number generator. The affiliation criterion was computed by finding the minimum Frobenius norm between a point and each of the *centres*. The epsilon distance used to evaluate the convergence criterion of the *centres* was set to 1×10^{-9}.

The algorithm used to taskify and parallelise the k-means application is the same as the one used in [7] and shortly presented in a simplified version in Algorithm 6 for the record. The distributed part of the algorithm is the call to the **`cluster_partial_sum`** function. The reduction of the array of *centres* into the accumulation variable $\texttt{centres}_{acc}$ is serialised on the master side, as well as the concatenation of the *label* lists. The function returns the computed *centres* along with the association between *points* and *centres*, data carried by *labels*.

The algorithm executes three main steps. First, **`generate_fragments`** randomly generates the points of all *fragments* which will be distributed to the different tasks. The centres are initially common for all fragments, although each application of **`cluster_partial_sum`** will modify them. **`cluster_partial_sum`** is used in order to compute the *labels* for each point and the *centre*'s position based on the clustered points. The position of one cluster *centre* is the barycentre of the *fragment*'s points belonging to this cluster. Finally, the *centres*' positions are reduced by calculating the means of the *centres* coordinates, for each *centre*. Hence, $\texttt{centres}_{acc}$ contains the mean of the means of each fragment clusters. If between two iterations,

Algorithm 6 Main loop for distributed k-means application.

Input

`N`	Total number of points to cluster
`k`	Number of clusters
`f`	Number of fragments
`max_iterations`	Maximum number of iteration
ε	Convergence criterion

Output

$\texttt{centres}_{acc}$	Vector of centres
`labels`	Association between points and centres

1: $(\texttt{ctrs}, \texttt{lbls}) \leftarrow$ **generate_centres**(`k`)
2: `fragments` $\leftarrow$ **generate_fragments**(`N`, `f`)
3: **for** `iter` $\leftarrow 1$ **to** `max_iterations` **do**
4: $\quad \texttt{centres}_{acc} \leftarrow \overrightarrow{0}$
5: $\quad$ `labels` $\leftarrow$ **empty_list**()
6: $\quad \texttt{centres}_{old} \leftarrow \texttt{ctrs}$
7: $\quad$ **for all** `frg` $\in$ `fragments` **do**
8: $\quad\quad$ **cluster_partial_sum**(`frg`, `ctrs`, `lbls`)
9: $\quad\quad$ **list_append**(`labels`, `lbls`)
10: $\quad\quad \texttt{centres}_{acc} \leftarrow \texttt{centres}_{acc} + {}^{1}/_{\texttt{f}} \cdot \texttt{ctrs}$
11: $\quad\quad \texttt{ctrs} \leftarrow \texttt{centres}_{old}$
12: $\quad$ **end for**
13: $\quad$ **if** $\|\texttt{centres}_{acc} - \texttt{centres}_{old}\| > \varepsilon$ **then**
14: $\quad\quad \texttt{ctrs} \leftarrow \texttt{centres}_{acc}$
15: $\quad$ **else**
$\quad\quad$ ▷ Early exit when convergence criterion is met.
16: $\quad\quad$ **break**
17: $\quad$ **end if**
18: **end for**
19: **return** $(\texttt{centres}_{acc}, \texttt{lbls})$

the difference between the previous and the new position of all *centres* is below the threshold ε, the function returns without executing the remaining iterations.

The function **`cluster_partial_sum`** takes as parameters one *fragment*, the corresponding *labels* and the redefined *centres*. Only the *fragment* is defined as *recurrent read-only* and hence is loaded into shared-memory. Contrary to the algorithm presented, the *labels* are modified as a side effect while the updated *centres* are returned by the function. However, because the execution of the function is realised by a worker, and as *labels* and *centres* are collections, their input and output are handled with the data being serialised to disk, waiting for the master to deserialise them, due to COMPSs behaviour. The timer starts just after the generation of the *fragments* and finishes when the main loop finishes (either by running out of iteration or by meeting the criterion of convergence), after all the *labels* have been gathered.

Table 5.1 presents the raw results from the application. Each case was run at least 50 times both with and without the use of shared-memory. White lines (cases 6, 8, 15 to 18 and 20) present test-cases where the difference in means of the timing in the method that uses the shared-memory is not significantly different from 0, meaning that there are no statistically significant differences between the timings due to standard variability (p-value <0.05). Red lines (case 19) are for cases where the difference in means is significantly greater than 0, meaning that there is a negative effect in performance when using the shared-memory that is likely not due to standard variability. Blue lines show cases where the performance improvement is statistically significant.

Cases 11 to 14 show that changing the number of iterations does not proportionally affect the gain in performance. The benchmark execution time is already dominated by the execution time of the iteration, which performance is improved proportionally to the improvement of the data deserialisation part. Even the dummy-case 1 which finishes early after only 2 iterations shows a difference in means close to 10 %. The 2–5 % difference can be explained by the original sending of data to the workers whose cost

ID	number of points	max. iter.	dims	centres	time with	time without	95 % conf. inter. inferior & superior in seconds		in percentages (%)		p-value
1	4 194 304	20	64	1	79.42	86.54	−8.23	−6.02	−9.51 %	−6.95 %	1.167×10^{-20}
2	4 194 304	20	64	2	627.5	733.3	−112.7	−99.0	−15.37 %	−13.50 %	1.700×10^{-23}
3	4 194 304	20	64	3	760.3	870.3	−114.2	−105.7	−13.12 %	−12.15 %	2.596×10^{-70}
▷4	4 194 304	20	64	4	890.7	987.6	−102.5	−91.6	−10.38 %	−9.26 %	1.233×10^{-50}
5	4 194 304	20	64	5	1028	1138	−118	−104	−10.33 %	−9.13 %	1.612×10^{-53}
6	4 194 304	20	8	4	862.7	858.8	−3.3	11.2	−0.38 %	1.31 %	0.271 527 7
7	4 194 304	20	16	4	859.1	871.5	−18.6	−6.2	−2.14 %	−0.71 %	1.891×10^{-4}
8	4 194 304	20	32	4	873.1	872.8	−4.7	5.5	−0.54 %	0.63 %	0.883 804 2
▷4	4 194 304	20	64	4	890.7	987.6	−102.5	−91.5	−10.38 %	−9.26 %	1.233×10^{-50}
9	4 194 304	20	96	4	905	1331	−434	−419	−32.62 %	−31.46 %	1.950×10^{-89}
10	4 194 304	20	128	4	941	1585	−652	−637	−41.09 %	−40.19 %	2.134×10^{-86}
11	4 194 304	10	64	4	457.0	513.0	−59.4	−52.7	−11.59 %	−10.27 %	3.769×10^{-52}
▷4	4 194 304	20	64	4	890.7	987.6	−102.5	−91.5	−10.38 %	−9.26 %	1.233×10^{-50}
12	4 194 304	30	64	4	1333	1490	−180	−133	−12.14 %	−8.98 %	2.337×10^{-19}
13	4 194 304	40	64	4	1734	1956	−233	−209	−11.96 %	−10.73 %	5.183×10^{-56}
14	4 194 304	50	64	4	2181	2443	−276	−247	−11.32 %	−10.13 %	8.564×10^{-55}

Table 5.1: Raw results for K-Means clustering application.

ID	number of points	max. iter.	dims	centres	time with	time without	95 % conf. inter. inferior & superior in seconds		in percentages (%)		p-value
15	2048	20	64	4	6.647	6.684	−0.189	0.116	−2.83 %	1.73 %	0.631 266 2
16	32 768	20	64	4	13.85	14.16	−0.73	0.11	−5.12 %	0.76 %	0.143 467 4
17	262 144	20	64	4	60.33	60.47	−0.53	0.25	−0.87 %	0.41 %	0.480 262 9
18	524 288	20	64	4	117.4	120.1	−12.5	7.1	−10.42 %	5.88 %	0.580 506 7
19	1 048 576	20	64	4	229.9	225.6	3.0	5.6	1.33 %	2.49 %	3.280×10^{-9}
20	2 097 152	20	64	4	448.8	447.7	−0.7	2.9	−0.15 %	0.65 %	0.219 859 5
▷4	4 194 304	20	64	4	890.7	987.6	−102.5	−91.5	−10.38 %	−9.26 %	1.233×10^{-50}
21	8 388 608	20	64	4	1801	2409	−619	−596	−25.7 %	−24.75 %	4.740×10^{-94}
22	16 777 216	20	64	4	3492	4835	−1368	−1319	−28.29 %	−27.29 %	7.419×10^{-121}
23	33 554 432	20	64	4	7076	9737	−2710	−2612	−27.90 %	−26.76 %	8.803×10^{-64}

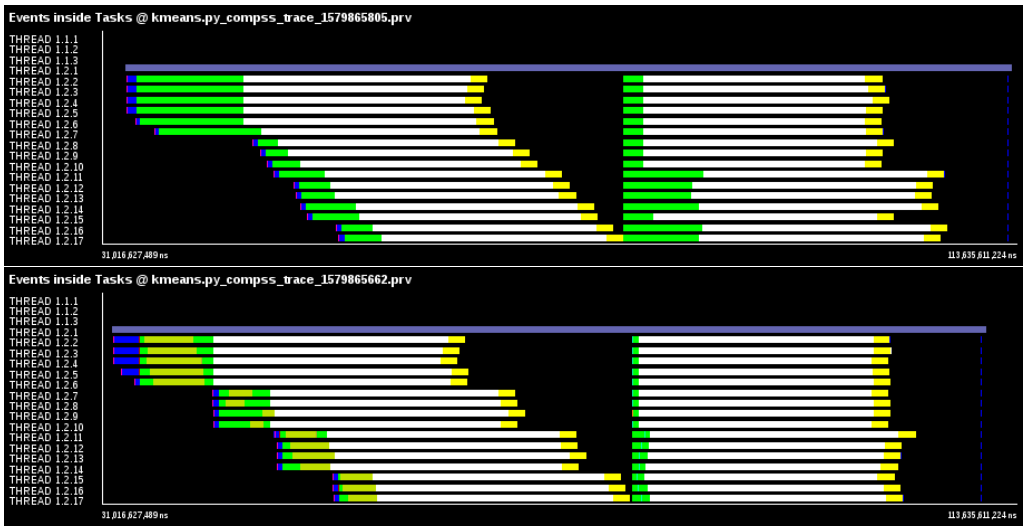
The maximum number of iterations is the number of iterations of the algorithm if no convergence between the *centres* happens first. The confidence interval correspond to the 95 % confidence interval of the difference in mean of each subgroup (*time with* or *time without*). Times are given in seconds. The difference in percentage is relative to the base time, i.e. time without the usage of shared-memory.

Table 5.1: Raw results for K-Means clustering application, continued.

is no longer negligible compared to the time to run the iterations.

The cases 6 to 10 and 15 to 23 show the impact of the amount of data to be loaded (by increasing either the number of points or the number of dimensions). In most of cases past some thresholds, the more data, the more improvement can be achieved. Also, the performance improvement seems to have a stronger scale with the number of dimensions than with the number of points. Thus, doubling the amount of data by doubling the number of points boosted the gain in performance from 9–10 % to 24–25 % while doubling the number of dimensions increased the gain to 40–41 %. One reason is that increasing the number of points also increases the number of *labels* to serialise and deserialise for each iteration, which impacts the performance negatively. It also appears that the gain in performance can be bounded as cases 22 and 23 both expose a gain around 25–28 %. However, this hypothesis could not be verified with the data that was gathered.

For cases 6, 8 and 15 to 20 the method to be used to load data does not seem to be very influential as the amount of data (below 64 MB per task) is too small to see any impactful improvement from reuse of memory. Hence the difference in means would only increase the variance of the timing distribution, participating in the noise of the measures. However, test case 19 shows that it can be disadvantageous to use shared-memory. The efficiency of the shared-memory execution path relies on the condition that the look-up through one table or the access to the system shared-memory is more efficient than accessing data on disk through standard NumPy deserialisation. Although it may be an artefact in the execution of the benchmarks, this test case shows that this condition is not necessarily always met, but the reason for this behaviour could not be determined for this set of parameters. In this underperforming case, the overhead still stays very low, below 2.5 %. The comparison between cases 6 to 10 and cases 15 to 20 show that the amount of data, especially the ratio of *read-only* memory over total memory to load is critical to finding cases where improvement can be reached using shared-memory. This ratio limits the growth in performance to be expected, hence, multiplying by a factor 4 the threshold before seeing a substantial gain in performance.



(a) Parameters are similar to test-case 1, convergence in 2 iterations. This figure illustrates a typical trace of the first two iterations of any test-case.

(b) Over the time of the application, reuse of data makes the cost of data serialisation insignificant. Parameters are similar to test-case 10.

Traces shown are from worker cores only. For each figure, top image shows the trace without shared memory while the bottom one uses shared memory. The colours are given chronologically, from left to right: lighter green segments represent the deserialisation of objects, darker green being the creation and population of the shared-memory segments; the large white segments in the middle represent the user code execution; yellow segments represent the serialisation of objects. As shown, not all objects are added to shared-memory, hence the deserialisations timing still appear on the traces shown in Figure 5.2-b, but most of it has been removed.

Figure 5.2: Traces of k-means PyCOMPSs application.

Figure 5.2 presents screenshots of postmortem visualisation of k-means application traces using the PARAVER [116] software. Figure 5.2-a shows that although the first iteration deserialisation gets slightly longer because of the loading of data to the shared-memory (light green at the origin of each segment), from the second iteration onward the deserialisation time is greatly reduced. The darker shade of green on the bottom trace of Figure 5.2-a at the beginning of each segment of the first column corresponds to the time spent loading the data into shared-memory.

As shown in Figure 5.2-b, the overhead that was required on the first iteration gets evened out by quick data recovery in shared-memory, and by even quicker data load from the internal dictionary during subsequent iterations.

5.4.2 Blocked Matrix Multiplication

The second case considered is a task-based version of the blocked 2D-matrix multiplication. The main algorithm is depicted in Algorithm 7. The data dependency expressed on the resulting matrix C is detected automatically by PyCOMPSs and managed with the COMPSs scheduler. Each matrix is composed by sub matrices indexed by their row and column numbers, starting at 0. All blocks are square and share the same dimensions, expressed in the amount of double precision floating point numerical values. During the experiments, the timer was started after the initialisation of the matrices, before the main multiplication loop. The call to the `matmul_block` function triggers a task creation that is executed by some worker process on the second node. Once all tasks were finished, after a synchronisation barrier, the timer was stopped. As expressed in the input parameters of Algorithm 7, only the two read-only input matrices A and B use the shared-memory capabilities.

For these experiments, two parameters varied. The number of blocks per matrix dimension and the number of elements per dimension of each block (reported as block size). The former influences the number of tasks generated in the application, while the latter modifies the amount of data

Algorithm 7 Main algorithm for blocked matrix multiplication.

Input

`dim`	Matrices dimension
A, B	Square 2D matrices, shared
C	Square 2D matrix, not shared, zero initialised

Output

C	Result matrix

```
1: for r ← 0 to dim − 1 do
2:     for c ← 0 to dim − 1 do
3:         for i ← 0 to dim − 1 do
4:             C_rc ← matmul_block(A_ri, B_ic, C_rc)
5:         end for
6:     end for
7: end for
```

to be loaded when deserialising a block. It also has a direct impact on the ratio $\frac{\text{user code execution time}}{\text{data and task management time}}$, which limits the proportion of code that can be improved with the addition to PyCOMPSs. As this experiment tries to show a reduction in the overhead induced by the programming model, increasing the number of elements too much can make the improvement disappear in comparison to the overall application time.

The preliminary parameter exploration showed that the most interesting performance test-cases included 8, 12, 16, 20 and 24 blocks in each dimension, with 128, 512, 1024, 2048 and 4096 doubles per dimension for each block. Each case was run 50 times with and 50 times without the usage of shared-memory. The different parameters tested and their results are reported in Table 5.2. The results exposed are similar as in Table 5.1, namely average timings with and without the extension, 95 % confidence interval of difference in mean and p-value. For a block size of 4096 elements, only the 8 blocks case gave results, as memory exhaustion led Python to raise exceptions when trying to execute larger cases. Thus, comparative tests could be run, but finer control over the memory allocation should make it possible to run larger cases.

From the 21 test-cases, 9 do not show any statistical difference in means when compared with a Welsh's t-test. 58 % of the remaining cases show an

ID	number of blocks	block size	time with shared-mem.	time without shared-mem.	95 % confidence interval inferior & superior in seconds		in percentage		p-value
1	8	128	8.434	8.360	−0.174	0.322	−2.08 %	3.85 %	0.555 681 8
2	12	128	18.143	18.495	−0.783	0.080	−4.24 %	0.43 %	0.108 780 0
3	16	128	43.624	42.093	0.034	3.027	0.08 %	7.19 %	0.045 101 7
4	20	128	78.447	77.561	−1.133	2.904	−1.46 %	3.74 %	0.386 067 1
5	24	128	137.690	136.848	−3.018	4.701	−2.21 %	3.44 %	0.665 251 6
6	8	512	18.287	17.343	0.088	1.800	0.51 %	10.38 %	0.031 097 6
7	12	512	51.527	48.920	0.717	4.496	1.46 %	9.19 %	7.366×10^{-3}
8	16	512	121.165	112.291	3.964	13.784	3.53 %	12.28 %	6.928×10^{-4}
9	20	512	233.181	211.867	12.811	29.818	6.05 %	14.07 %	2.969×10^{-6}
10	24	512	386.480	379.221	−5.643	20.161	−1.49 %	5.32 %	0.266 950 3
11	8	1024	46.353	52.348	−8.472	−3.519	−16.18 %	−6.72 %	8.376×10^{-6}
12	12	1024	151.787	160.587	−16.390	−1.210	−10.21 %	−0.75 %	0.023 540 3
13	16	1024	383.510	386.397	−7.897	2.122	−2.04 %	0.55 %	0.257 332 5
14	20	1024	741.622	757.913	−25.440	−7.143	−3.36 %	−0.94 %	5.524×10^{-4}
15	24	1024	1249.294	1222.476	−0.847	54.483	−0.07 %	4.46 %	0.057 242 6

Table 5.2: Raw results for blocked matrix multiplication.

ID	number of blocks	block size	time with shared-mem.	time without shared-mem.	95 % confidence interval inferior & superior in seconds		in percentage		p-value
16	8	2048	180.111	204.291	−31.049	−17.311	−15.20 %	−8.47 %	3.446×10^{-10}
17	12	2048	581.257	637.424	−76.866	−35.468	−12.06 %	−5.56 %	5.083×10^{-7}
18	16	2048	1404.155	1551.578	−203.842	−91.005	−13.14 %	−5.87 %	2.381×10^{-6}
19	20	2048	3003.337	3083.178	−249.923	90.241	−8.11 %	2.93 %	0.350 365 8
20	24	2048	4732.841	5034.617	−679.851	76.298	−13.50 %	1.52 %	0.111 333 2
21	8	4096	786.623	925.552	−154.124	−123.735	−16.65 %	−13.37 %	8.347×10^{-33}
22	12	4096	N/A	3219.718	—	—	—	—	—

The number of blocks and the block sizes are the same in each dimension as the matrices are square. Times are given in seconds. The boundaries of the 95 % confidence interval are for the difference in mean of each subgroup (time *with* or *without* shared-memory). The difference in percentage is relative to the base time, i.e. time without the usage of shared-memory.

Table 5.2: Raw results for blocked matrix multiplication, continued.

improvement in performance, ranging from <1 % to 16.65 % for improved cases. The worst penalty measured is an overhead of 14.07 % in case 9. For cases 1 to 10, the differences in mean of cases at issue show respectively 3.64 %, 5.4 %, 5.3 %, 7.9 % and 10 % of runtime increase when the application is being run with the shared-memory. The variability in the results can be explained from a lack of fine grain control over the tasks attribution. This reduces the direct reuse of previously deserialised matrices that are stored in the internal dictionary. The creation of a shared-memory array requires executing the memory allocation twice, once from file to memory, and once from memory to shared-memory, and needs the data to be written both times. Hence, this overhead can only overcome its cost with sufficient reuse of memory pages. It appears that bigger sizes lead to better results, as the loading from the disk can throttle the performance compared to memory mapping.

Future work is planned to include testing with regard to fine grain task scheduling in order to both reduce the variability of measurements and to try to define an optimal scheduling. As a matter of fact, a fine grain management of task scheduling that would prioritise tasks with disjoint data sets would improve performance as fewer shared-memory blocks would be rewritten in the case of data name conflicts. In the case of matrix multiplication, all blocks have to be used multiple times. On a single node, one optimised algorithm could be to first compute the blocks along the diagonal to maximise the number of arrays ready to be reused, and to minimise the number of arrays loaded twice to shared-memory. The workers could execute computing operations following the row, for example, in order to reuse the blocks deserialised from matrix A. After computing all operations where A was required, the worker could compute blocks from the column in order to reuse the blocks deserialised from matrix B.

Although two thirds of the cases show an improvement when using shared-memory, the selection of good candidates is a difficult issue for applications with a complex pattern of memory accesses. As an example, increasing the number of blocks increases the parallelism of the application and the number of times the data have to be used. But it also increases the

number of times the output matrices have to be serialised and deserialised. As shown in cases 11 to 14 and cases 16 to 18, for one given block size, the increase in the number of blocks decreases the gain in performance, from a maximum of 11.45 % (case 11) to a minimum of 2.15 % (case 14) on average for a block size of 1024 and from a maximum of 15.20 % (case 16) to a minimum of 8.81 % (case 17) on average for a block size of 2048, respectively.

5.5 Conclusion

This work presents the integration of an extension to the Python language features into the PyCOMPSs framework. Analogously to the original work, the code modification required to use this new trait is kept to a minimum in order to make code adaptation as easy as possible and provide a seamless integration into the parallel framework. For algorithms based on a high amount of reuse of data, such as k-means, performance has been shown to be improved by at least 10 % or unaffected by using shared-memory, under the conditions of experiment. For improved cases, the effect is amplified as the amount of data is increased, proportionally to the total amount of I/O, with an improvement of ≈40 % when increasing the number of point dimensions in the case of a k-means application. However, for a too small task granularity there can be an antagonistic, yet relatively limited effect. Nonetheless, the promising results based on read-only memory may lead to a possible evolution of the framework allowing a better usage of shared-memory when distributing tasks to different workers sharing the same node, to reduce even more the number of serialisations required, for applications behaving like the blocked matrix multiplication benchmark. Larger scale testing of the k-means application could be done for a very large dataset of points to see the integration with storage solutions such as Redis (as presented in [22]) to evaluate the behaviour in the case of unsupervised distributed applications and to verify whether the solution still provides an improvement of the general performance.

An additional outcome of this work is to display the eligibility of Python based applications and workflows to be executed with memory placement

tools and for their performance to be improved with standard memory-oriented optimisations. This second noteworthy result will lead to further research on integration with low level memory management on the strength of the assumptions provided by a higher-level language.

Chapter 6

Conclusion

Contents

6.1 Summary of the contributions

Modern supercomputer architectures have both complicated in-node memory systems, and memory which is distributed over the inter-node network. Research into efficient data management is therefore much more complex as the diversity of memory increases. In addition, abstractions like NUMA distance or NUMA domains are being broken as the memory characteristics are getting more diverse. Moreover, the diversity of environments, computing systems and memory systems requires the development of new portable methods to reduce the effort called for maintaining code bases with very long lifetime.

The primary approach to resolving this issue is to use languages that are versatile enough to provide new programming models hoping that may be included eventually in the standard. This however lacks some portability as it depends on compiler support for language evolution, and limits the languages that can be used for the development of applications. Another approach is to rely on frameworks which better support different languages, but is still being limited by the compiling tool chain chosen. Section 2.2 summarised the different solutions at a memory design level, at an OS level and at the programming language level.

A corollary issue to the multiplicity of memory systems is the increased difficulty of memory partitioning management. The disaggregation implicates managing multiple source of data with some being unreachable (e.g. a CPU cannot access data located on GPU embedded HBM). Hence the selection of data by intended usage and destination enables an effective data management. The ASPEN algorithm improves the state-of-the-art for computing complete redistribution of block-cyclic data across two distinct N-dimensional grids (see Section 3.4.5). The work revolving around the development of this algorithm was allotted as follows:

- The development of the algorithm was done as a collaborative work between Adrian Tate and the author;
- The implementation of the ASPEN algorithm for benchmarking purpose was done by the author;

- The implementation of the different algorithms based on the algorithmic description for comparison purpose, as presented in Section 3.5, was done by the author;
- The implementation and adjustments that were required to include the ASPEN algorithm into the UDJ library was done by the author.

The plurality of memory systems comes with dedicated APIs for the selection of data sets or subsets to be allocated or copied in the different memory tiers. This implicates rewriting portions of code when changing library or memory provider as the memory allocation interface may changed between libraries (see Section 4.4). This observation led to the conclusion that providing a unique API, shared between the different memory providers, would be beneficial and reduce the amount of work required by the community. Therefore, the first objective of Mamba is to provide a simple compatibility layer for several libraries and a framework for easier collaboration between them. The collaborative work has been done with the following task division:

- The overall library and its API has been drafted as a collaboration between Adrian Tate, Tim Dykes and the author;
- The tiling management and array abstraction was the result of Tim Dykes' work, with minor contributions from the author to ensure compatibility with the memory management part of the library;
- The memory managements and memory operations part of the library, along with the topology discovery were implemented by the author;
- The author developed and ran the benchmarks used in Section 4.4.

These approaches are designed to suit low level languages as used in numerical simulation, e.g. C, C++ or Fortran to name the more broadly used ones. However, many domain scientists do not rely on these and they would benefit from being provided with the tools that are simpler to integrate. For example, Python has been gaining traction for its ease of use and versatility. Yet, support for precise memory management is lacking as, in our case, the support for inter-process memory sharing was lacking flexibility. Henceforth, an external library was developed to provide sufficient support for the distributed tasking framework PyCOMPSs (see

Section 5.3.2.1). This provided a first glimpse into the design of the memory management in a high-level language, and an occasion to study the extent of memory management that could be provided to them. This author provided the following contributions:

- The Python extension was based on Mathieu Mirmont's work; the author extended it to support conflicting names for arrays, and a copy-constructor for the objects;
- The inclusion of the Python extension as part of the PyCOMPSs framework and the coupling between the serialisation and deserialisation processes was realised by the author;
- The support for the new decorator, the addition of the internal dictionary and the adaptation of benchmarks were done by the author.

6.2 Future work

6.2.1 Architecture evolutions and studies

The recent evolution of memory designs are shifting away from having one centralised memory to provide data to both CPUs and GPUs. The need for performant DRAM in HPC is decreasing as the amount of HBM on package increases. This trend seems to be confirmed by the 32 GB of HBM2 on Fujitsu A64FX processors, which is reminiscent of the MCDRAM used by Intel in the KNL system. The technology allows the stacking or up-to eight DRAM dice, next generation of processors could provide even more HBM2.

However, the inclusion of this technology seems to lead to the disappearance of the L3 cache level. This implicates a latency penalty when accessing data as the L2 cache is sensibly smaller. On the other hand, the inclusion of network interconnects in the processor package in addition to the standard PCIe ports may lead to the development of DRAM-over-network.

Moreover, GPU to GPU, GPU to PCIe card and GPU to GPU communications can also bypass the central memory for the data exchanges. This means the tree representation of the memory system does not fit anymore as some subset of the systems are now autonomous clusters for computation

and communication. The NUMA model which originally referred to a computing unit with some homogeneously accessed memory is being dropped, as NUMA nodes may now refer only to memory. The industry standard `hwloc` is already shifting to a graph model, with two kinds of nodes (*initiators* and *targets*) and edges with some attributes (e.g. latency, bandwidth).

Provided the aforementioned possible evolution of the memory organisation and given the closer inclusion of network communication, new memory attributes may arise to describe the system and take better data placement decisions. For example, given the sub-clusters that are appearing, it may be relevant to consider the *accessibility* of a memory system. Widely accessible memory (the central main memory or storage, for example) does not restrict the access to the data from any device, but it increases the complexity of the coherency management. GPU located memory forbids the CPU to access it, and requires a copy or a synchronisation point for operations not available on GPUs.

6.2.2 Data selection and locality

As pointed out in [24], one draw back of the general redistribution method is that it does not consider the network capabilities nor the data locality. Further research may include consideration of the block sizes to optimize the network utilisation.

Moreover, applications like distributed object storage may benefit from an efficient redistribution algorithm. It may help creating distributed storage systems that can automatically redistribute itself when growing in case of an increased demand. Only the dimensions of buckets of metadata would be required to compute which system contains the information requested to retrieve it.

6.2.3 Data movement and code portability

Providing portable memory management is especially complicated when memory classes are identified by NUMA node ids, which are not necessarily consistent between machines, or via the usage of a translation symbolic

name (as we did in Mamba with `MMB_HBM`, for example). It may be preferable to describe the memory needs directly as priority between attributes, which is more portable, as the requirement may or may not be met, depending on the hardware availability. However, setting a hard threshold to discriminate between memories can be complicated. Actually, a user may find acceptable to give a tolerance to the decided threshold, as it may be more convenient to expose two memory targets as one when they both expose similar characteristics.

6.2.4 Memory management for high-level languages

In Python, the memory management is handled by the interpreter. Although some libraries start providing support for heterogeneous computing systems, they are mostly relying on the system decision for heterogeneous memory management. Optimised libraries such as NumPy provide good performance for numerical applications by providing a good locality (data are densely stored in arrays), and provide high level operations (e.g. dot product). This high-level vision of the algorithm could provide hints for where to store the data. Furthermore, an operation like dot product which exposes regular accesses to memory can be implemented taking it in consideration, in addition to cache line sizes to improve performance. Interfacing a library like Mamba with the Python language could provide a way to do so, notably by using the Mamba tiling mechanisms.

Glossary

3D XPoint memory Pronounced *3D cross-point*, NVM technology developped jointly by Intel and Micron. 25, 26, 32, 34

Advanced Configuration and Power Interface Vendor provided information used in OS-directed configuration and power management. 147

Application Programming Interface A computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data format that should be used, the conventions to follow, etc. 147

Argonne Memory Library Library develop by ANL as building block for memory management. 147

Argonne National Laboratory A science and engineering research national laboratory operated by University of Chicago Argonne LLC. 42, 147

Artificial Intelligence Area of Computer Science aiming at making possible for machines to learn from experience, adjust to new inputs and perform human-like tasks. Using different technologies, computers can be trained to accomplish specific tasks by processing large amounts of data and recognising patterns in the data. 100, 147

Atomicity, Consistency, Isolation, Durability Properties of transactional storage class systems. The respect of these properties is in-

tended to guarantee validity even in the event of errors, power failures, etc. 28, 147

Bandwidth Throughput of memory. This is evaluated in amount of bytes transferred per second. 2, 15, 16, 23, 24, 29–32, 38, 47, 50, 130

Bank Circuit containing array of bit cells allowing a word-wide granularity independent access to the information stored in memory. 13

Big Data Field that analyses sets of data too complex or too large to be otherwise handled by traditionnal data analysis techniques and technologies. 2, 100

Burst Buffer A fast and intermediate storage layer positioned between the computing system and the network. 2, 29, 34

Bus General name for a computer communication interface that transfers data between different components. 13, 14, 20, 34, 135, 136, 139, 141, 143

Central Processing Unit Integrated electronic circuits in a computer that is responsible for performing arithmetic, logic, controlling, and I/O operations specified by the instructions in the program. 147

Channel Part of memory architecture allowing for better performance. One channel corresponds to the connexion between the memory controller and the memory module. Multiple channel-architectures allow for as many concurrent access to the different memory modules, providing more bandwidth. 14, 15, 23

COMP Superscalar A framework which aims to ease the development and execution of parallel applications for distributed infrastructures, such as Clusters, Clouds and containerised platforms. 147

Cori NERSC's largest supercomputer, which ranked 5^{th} on the TOP500 in November 2016. 30

Data Analytics Data analytics is the process of examining data sets in order to draw conclusions about the information they contain, increasingly with the aid of specialised systems and softwares. 55

Data Parallel C++ Single source programming model based on SYCL for distributed computation on host and devices. 44, 148

Direct Memory Access Direct reading and writing access to the main memory system, outside of CPU I/O. 139, 148

Dual Data Rate Characterises a synchronous data bus that transfers data both the raising and he failing edge of a clock signal. 14, 148

Dual in-line Memory Module A kind of memory module that comprises a series of DRAM memory integrated circuits. While the contacts on SIMMs on both sides are redundant, DIMMs have separate electrical contacts on each side of the module. 12, 14, 148

Dynamic Random-Access Memory A type of semiconductor memory that stores each bytes of data. This memory is said volatile. 22, 148

Embedded Dynamic Random-Access Memory DRAM used embedded on package, usually as L4 cache. 148

Ethernet Ethernet is a family of computer networking technologies widely used in homes and industries. Its standard is defined by the norm ISO/IEC/IEEE 8802.3. 142

Expansion Bus Connecting bus used to access a printed circuit board that can be inserted into an electrical connector to add functionality to a computer system. 141

ExaFLOP/s 10^{18} FLOP/s (Floating-point Operations Per Second). 2, 135

Exascale The capacity for a computing system to perform at least one ExaFLOP/s. 2, 16, 55, 74, 77, 78

Flash Flash memory is an electronic (solid-state) non-volatile computer memory storage medium that can be electrically erased and reprogrammed. 21, 25–27, 32, 140

Floating-point Operations Per Second Metric used to evaluate the performance of computing systems. It can either be used for *peak performance* or for *sustained performance*. 135, 141, 144, 148

Front-Side Bus Data bus connecting the CPU to the memory controller. 14, 148

Gigabyte 10^9 bytes. 110, 148

Global Interpreter Lock For Python, it is a mutex that allows only one thread to hold the control of the interpreter, ensuring the stability of the internal state of the interpreter. 102, 104, 148

Graphics Dual Data Rate DDR specialised for GPU memory as opposed to the plain DDR which is used for general purpose memory. 15, 148

Graphics Processing Unit Specialised processing unit originally dedicated to process images. On modern computers, the highly parallel computation power provided by those devices may be used to accelerate computations. 148

Hard disk drive Computer storage solution based on a fixed or spinning disks using magnetic characteristics to store and retrieve data. 149

Heterogeneous Computing Interface for Portability Vendor-neutral C++ programming model for GPUs. 149

High Bandwidth Memory 3D stacked memory, generally on package, providing a data throughput orders of magnitude superior to standard DRAM. 23, 149

High-Performance Computing Area of computer science where the applications require a large amount of computing power over various periods of time ranging from hours to years. 2, 100, 149

High-Performance Fortran Extension of Fortran 90 that aims at helping the support of parallel computing in Fortran. 58, 149

Hybrid Memory Cube First 3D organisation of memory cells. 23, 149

InfiniBand Networking standard used in HPC with high throughput and very low latency. 14, 29, 38, 142, 149

Input/Output Communication between one computing device and another computing device, memory or *the outside world*, i.e. input or output device, e.g. the keyboard or the computer monitor. The data going toward the main computing device is called *input*, and the data coming from the main computing device is called *output*. 106, 149

Joint Electron Device Engineering Council JEDEC Solid State Technology Association is an independent semiconductor engineering trade organisation and standardisation body composed of manufacturers and suppliers of the microelectronics industry. JEDEC's collaborative efforts ensure product interoperability, benefiting the industry and ultimately consumers by decreasing time-to-market and reducing product development costs. 149

Last Level Cache Generally refers to the further, in-package, cache level of the processor. It usually is the biggest although it is also the highest latency. The LLC is often shared between the different cores of the processor. 36, 149

Latency Time delay between the request of a random byte or word in *memory* and its availability in the processor. 2, 9, 16, 17, 29, 32, 47, 50, 130

Local Data Descriptor Set of ranks and dimensions describing a regular distribution of data across multiple agents is used to compute local or global indices of the scattered data. 56, 59, 149

Magnetoresistive Random-Access Memory A type of NVRAM using known magnetic effects to retain data. 150

Multi-Channel Dynamic Random-Access Memory Architecture for Intel Xeon Phi Knights Landing (KNL) HBM memory technology. 24, 150

Mean Time Between Failures The predicted elapsed time between inherent failures during normal system operation. For example, a failure may be an accumulation of random bit flips putting the program in an unstable state, or a state normally not reachable under normal execution. The term is used for repairable systems, while mean time to failure (MTTF) denotes the expected time to failure for a non-repairable system. 30, 150

Memory Management Unit Controller primarily performing the translation of virtual memory addresses to physical addresses, but also handles memory protection, cache control and bus arbitration. 150

Multilevel Cell Memory cell that can be sampled with enough multiple states to encore more than one bit of data. 27, 150

Multiple Instruction, Multiple Data Technique employed to achieve parallelism relying on multiple independent processing units executing instruction asynchronously. Supercomputer are an example of such technique of parallelisation. 150

Mutex Short for *mutual exclusion*, it is a synchronisation tool that allows an unique execution thread to access a critical section at once. 102, 136

National Energy Research Scientific Computing Center High-performance computing user facility operated by Lawrence Berkeley National Laboratory for the United States Department of Energy, Office of Science. 150

Network Interface Card Extension card, often connected to the PCIe. This card manages the interactions with the network, lowering the number of interruptions for the processor and allowing for the network to communicate in parallel of the computation. Those cards can also provide Direct Memory Access. 150

Network-Attached Storage Computer data storage server connected to a computer network providing data access to a heterogeneous group of clients. 29, 150

Non-Uniform Memory Access Memory design applied to computer with multiple processors where the speed to access memory depends on the location of the processor. 16, 150

Non-Volatile Opposite of volatile. 20–22, 27–29, 136, 139–141, 145, *see* Persistency, Volatile & Volatility

Non-Volatile Dual in-line Memory Module DIMM which memory is non-volatile. 150

Non-Volatile Memory Express High-performance interface specification for accessing NVM devices attached via PCIe bus. 151

Non-Volatile Memory A type of persistent memory technology used for computation. 25, 28, 33, 151, 168

Non-Volatile Random-Access Memory Generic denomination for any kind of non-volatile Random-Access Memory system used for computations. 150, *see* PMM

Not-AND Logic-gate producing the complement to a AND logic-gate. It acts like a boolean function which produce a 1 in output if any of the

input is 1, and 0 otherwise. This technology is used to create Flash non-volatile memory. 150

Not-OR Logic-gate producing the complement to a OR logic-gate. It acts like a boolean function which produce a 0 in output if any of the input is 1, and 1 otherwise. This technology is used to create Flash NVM. 150

NumPy An open source project aiming to enable numerical computing with Python. It was created in 2005, building on the early work of the Numerical and Numarray libraries. 3, 100–102, 106, 108, 109, 118, 131

Open Computing Language An open standard for writing code that runs across heterogeneous platforms including CPUs, GPUs, and other processors. 151

Open Multi-Processing Multi-platform shared-memory multiprocessing programming API for in C, C++, and Fortran languages. 151

Operating System A system software that manages computer hardware, software resources and provides common services for computer programs. 151

Package In the context of manufactured electronics, it refers to the casing containing one or more discrete semiconductor devices or integrated circuits. In the case of processors, this term denotes the association of the processing units, with their associated memories and some of the interfaces with external systems. 10

Parallel File System Storage system allowing virtually-unique file system spanned across multiple storage points enabling high-performance, scalable, concurrent, parallel accesses to files presented in a globally shared namespace. 23, 151

Partitioned Global Address Space In computer science, it refers to a parallel programming model. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process, thread, or processing element. 102, 151

Peripheral Component Interconnect Express A standard expansion bus providing high-speed, generally used to interface for personal computers' graphics cards, hard drives, SSDs, Wi-Fi and Ethernet hardware connections. 151

Persistency Inherent or expected characteristic of information to be retain without alteration over time. Persistent memories are said non-volatile. 2, 16, 143, *see* Volatile, Volatility & Non-Volatile

Persistent Memory Module DIMM of persistent memory. 26, 151, *see* NVRAM

PetaFLOP/s 10^{12} FLOP/s (Floating-point Operations Per Second). 31, 141

Petascale The capacity for a computing system to perform at least one PetaFLOP/s. 31

Phase-Change Memory Type of NVM using the physical capacity of chalcogenide glass to change their molecular structure when applied current. 27, 151, *see* PCRAM

Python Interpreted high-level, general-purpose programming language particularly used for scientific application and AI. 3–6, 50, 97, 100–109, 121, 125, 129, 131, 136

Python COMP Superscalar Python binding for the COMPSs framework. 151

Quad Data Rate Characterise a synchronous data bus similar in principle to DDR, but with a second clock signal being 90° out of phase from the first signal. Each raising and falling edges of each signal

provokes a data transfer, allowing a higher throughput than DDR. 14, 151

Radeon Open Compute AMD's open software ecosystem for accelerated compute. 152

Random-Access Memory Main memory used in computers to execute tasks. This memory is said *Random-Access* as any address can be directly accessed for both reading of writing operations, in almost the same amount of time irrespective of the physical location of data inside the memory. 31, 139, 151

Relatively Prime Two integers are relatively prime is they have no common factor. 59

Remote Direct Memory Access In computer science, it is a direct memory access from the memory of one computer into that of another without involving either one's operating system. This permits high-throughput, low-latency networking, which is especially useful in massively parallel computer clusters. 152

Resistive Random-Access Memory Computer memory technology that works by changing the resistance across a dielectric solid-state material, often referred to as a memristor or RRAM. 27, 152

Remote Direct Memory Access over Converged Ethernet Communication protocol based on IB (InfiniBand) transport packet over Ethernet. 152

ScaLAPACK The ScaLAPACK (or Scalable LAPACK) library includes a subset of LAPACK routines redesigned for distributed memory MIMD parallel computers. 4, 63, 69, 70

Scratch space Temporary storage used to buffer data until the computation it is required for can start. 29

Shared Virtual Memory Virtual addressing space for memory spanned across multiple memory systems. 40, 153

Single Data Rate Characterises a synchronous data bus that transfers data on either exclusively the raising or the failing edge of a clock signal. 14, 152

Single in-line Memory Module A type of memory module containing random-access memory used in computers from the early 1980s to the late 1990s. It differentiates with the DIMM in that only one side or the module is providing memory. 152

Single-Intruction-Multiple-Data Technique of parallelisation based on applying one single operation to a set of data, generally of the same kind, all at the same time. Vectorisation is a application of such a parallelisation technique. 152

Solid-State Drive Devices that uses to store data persistently, typically using flash memory. It usually is used as secondary storage as it usually stores data that cannot be kept in main memory (RAM) for a lack of physical space on the device or for persistency reasons. 153

Spin-Transfer Torque Magnetoresistive Random-Access Memory Technology using the orientation of a magnetic layer in a magnetic tunnel junction to retain bit state. This is a studied candidate for NVRAM. 27, 153

Static Random-Access Memory A type of semiconductor memory that stores each bit of data using a bistable latching circuitry. This memory is said volatile although while being powered this memory exposes data remanence, sparing the need for refresh contrary to DRAM. 22, 152

Storage Networking Industry Association The SNIA is a non-profit global organisation dedicated to developing standards and education programs to advance storage and information technology. 152

Stride In the context of array redistribution, the stride represents the minimum size containing a complete pattern of data exchange. 63, 65

SYCL Cross-platform abstraction layer that enables code for heterogeneous processors to be written using standard ISO C++ with the host and kernel code for an application contained in the same source file.. 44, 45, 51, 76, 135

Symmetric Hierarchical Memory From Cray Research's *shared memory* library, it is a family of parallel programming libraries, providing one-sided, RDMA, parallel-processing interfaces for low-latency distributed-memory supercomputers. The SHMEM acronym was subsequently reverse engineered to mean *Symmetric Hierarchical MEMory*. 152

Symmetric Multiprocessing Architecture of hardware and software for multiprocessor computer where two or more identical processors are connected to a single, shared main memory, have full access to all input and output devices, and are controlled by a single operating system instance that treats all processors equally, reserving none for special purposes. 102, 152

Synchronous Dynamic Random-Access Memory A type of DRAM where the access operations are synchronised on a clock signal provided externally. 12, 152

TeraFLOP/s 10^{15} FLOP/s (Floating-point Operations Per Second). 31

Terabytes 10^{12} bytes. 153

TOP500 Ranking and details of the 500 most powerful non-distributed computer systems in the world. The project was started in 1993 and publishes an updated list of the supercomputers twice a year (June and November, respectively during ISC and SC, two HPC-oriented conferences). 31, 134

Volatile Expression of the volatility characteristic of the memory. All data stored in a volatile memory is lost on the event of losing power. The memory that does not lose its data in the event of losing power is said non-volatile. 135, 139, 143, *see* Volatility & Non-Volatile

Volatility Characteristic expressing the capacity for a memory to retain stored information permanently, across either shut-down or power-loss. 16, 50, 145, *see* Volatile & Persistency

Write Amplification Undesirable effect for flash based memory systems that prematurely exhaust memory cells due to the coarsened granularity of the writing operation that imposes the erasure of unmodified memory cells. 34

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 28, *Glossary:* Atomicity, Consistency, Isolation, Durability

ACPI Advanced Configuration and Power Interface. 47, *Glossary:* Advanced Configuration and Power Interface

AI Artificial Intelligence. 2, 55, 100, 141, *Glossary:* Artificial Intelligence

AMD Advance Micro Devices. 46, 51, 142

AML Argonne Memory Library. 42, 51, 78, *Glossary:* Argonne Memory Library

ANL Argonne National Laboratory. 31, 42, 133, *Glossary:* Argonne National Laboratory

API Application Programming Interface. 3, 28, 34, 35, 38, 44, 50, 76–79, 82, 84–86, 88–90, 97, 101–103, 106, 108–110, 128, 129, 140, *Glossary:* Application Programming Interface

C/R Checkpoint-Restart. 31

COMPSs COMP Superscalar. 101, 103–106, 115, 120, 141, *Glossary:* COMP Superscalar

CPU Central Processing Unit. 2, 33, 37, 38, 40, 42, 47, 51, 77–79, 89, 128–130, 135, 136, 140, 157, *Glossary:* Central Processing Unit

DRAM Dynamic Random-Access Memory. 2, 11–28, 31–34, 36–38, 40–43, 48, 51, 82, 88, 129, 130, 135, 136, 143, 144, *Glossary:* Dynamic Random-Access Memory

DDR Dual Data Rate. 14, 15, 19, 24–27, 136, 141, 142, *Glossary:* Dual Data Rate

DIMM dual in-line memory module. 12, 14, 17–19, 24, 32, 82, 135, 139, 141, 143, *Glossary:* Dual in-line Memory Module

DMA Direct Memory Access. 29, 41, 48, 139, *Glossary:* Direct Memory Access

DPC++ Data Parallel C++. 44, 51, *Glossary:* Data Parallel C++

eDRAM Embedded Dynamic Random-Access Memory. 12, 27, *Glossary:* Embedded Dynamic Random-Access Memory

ESS Elastic Storage Support. 29

FLOP/s Floating-point Operations Per Second. 135, 141, 144, *Glossary:* Floating-point Operations Per Second

FPGA Field-programmable gate array. 2

FSB Front-Side Bus. 14, *Glossary:* Front-Side Bus

GDDR Graphics Dual Data Rate. 15, 42, 43, *Glossary:* Graphics Dual Data Rate

GB Gigabyte. 11, 19, 23, 24, 26, 27, 81, 89, 105, 129, *Glossary:* Gigabyte

GIL Global Interpreter Lock. 102, 104, *Glossary:* Global Interpreter Lock

GPU Graphics Processing Unit. 2, 15, 18, 34, 38, 40, 42, 46, 51, 76–79, 82, 84, 86, 88–90, 128–130, 136, 140, *Glossary:* Graphics Processing Unit

HBM High Bandwidth Memory. 12, 16, 17, 23, 24, 42, 43, 46, 47, 51, 85, 88, 128, 129, 138, *Glossary:* High Bandwidth Memory

HDD Hard disk drive. 26, 31, *Glossary:* Hard disk drive

HIP Heterogeneous Computing Interface for Portability. 43, 45, 46, 51, 76, 84, *Glossary:* Heterogeneous Computing Interface for Portability

HMC Hybrid Memory Cube. 23, 24, *Glossary:* Hybrid Memory Cube

HPC High-Performance Computing. 2, 4, 8, 23, 28–30, 42, 47, 54, 55, 72, 97, 100, 129, 137, 144, *Glossary:* High-Performance Computing

HPDA High-Performance Data Analysis. 2

HPF High-Performance Fortran. 58, *Glossary:* High-Performance Fortran

I/O Input/Output. 3, 23, 25, 26, 29, 30, 32, 106, 112, 125, 134, 135, *Glossary:* Input/Output

IB InfiniBand. 14, 29, 38, 142, *Glossary:* InfiniBand

IPU Intelligence Processing Unit. 2

JEDEC Joint Electron Device Engineering Council. 15, 23, 25, 27, 29, 32, *Glossary:* Joint Electron Device Engineering Council

LANL Los Alamos National Laboratory. 31, 43

LAPACK Linear Algebra Package. 142

LBNL Lawrence Berkeley National Laboratory. 139

LDD Local Data Descriptor. 56, 59, *Glossary:* Local Data Descriptor

LLC Last Level Cache. 23, 24, 36, *Glossary:* Last Level Cache

LLNL Lawrence Livermore National Laboratory. 31, 42–44

MCDRAM Multi-Channel Dynamic Random-Access Memory. 16, 24, 37, 47, 88, 129, *Glossary:* Multi-Channel Dynamic Random-Access Memory

MIMD Multiple Instruction, Multiple Data. 30, 142, *Glossary:* Multiple Instruction, Multiple Data

MLC Multilevel Cell. 27, 28, *Glossary:* Multilevel Cell

MMU Memory Management Unit. 28, 42, *Glossary:* Memory Management Unit

MRAM Magnetoresistive Random-Access Memory. *Glossary:* Magnetoresistive Random-Access Memory

MTBF Mean Time Between Failures. 30, *Glossary:* Mean Time Between Failures

NAND Not-AND. 25–27, *Glossary:* Not-AND

NAS Network-Attached Storage. 29, 34, *Glossary:* Network-Attached Storage

NERSC National Energy Research Scientific Computing Center. 30, 134, *Glossary:* National Energy Research Scientific Computing Center

NIC Network Interface Card. 29, *Glossary:* Network Interface Card

NOR Not-OR. 26, *Glossary:* Not-OR

NUMA Non-Uniform Memory Access. 3, 11, 16, 33, 37, 42, 47, 74, 88, 90, 128, 130, 131, *Glossary:* Non-Uniform Memory Access

NVRAM Non-Volatile Random-Access Memory. 31, 138, 143, *Glossary:* Non-Volatile Random-Access Memory

NVDIMM Non-Volatile Dual in-line Memory Module. 11, 17, 25, 31–33, 43, 51, 82, 86, 88, *Glossary:* Non-Volatile Dual in-line Memory Module

NVM Non-Volatile Memory. 21, 25, 26, 28, 33, 133, 139–141, 168, *Glossary:* Non-Volatile Memory

NVMe Non-Volatile Memory Express. *Glossary:* Non-Volatile Memory Express

OpenCL Open Computing Language. 38–40, 44, 45, 48, 51, 76, 89, *Glossary:* Open Computing Language

OpenMP Open Multi-Processing. 38, 39, 45, 48, 51, 76–78, 101, 102, *Glossary:* Open Multi-Processing

ORNL Oak Ridge National Laboratory. 43, 44

OS Operating System. 37, 128, 133, *Glossary:* Operating System

PCIe Peripheral Component Interconnect Express. 14, 32, 41, 130, 139, *Glossary:* Peripheral Component Interconnect Express

PCM Phase-Change Memory. 26, 27, *Glossary:* Phase-Change Memory

PCRAM Phase-Change Random-Access Memory. 31, *see* PCM

PFS Parallel File System. 30, *Glossary:* Parallel File System

PGAS Partitioned Global Address Space. 102, *Glossary:* Partitioned Global Address Space

PMM Persistent Memory Module. 29, 32–34, 84, *Glossary:* Persistent Memory Module

PyCOMPSs Python COMP Superscalar. 3, 5, 100, 101, 103–107, 112, 119–121, 125, 129, *Glossary:* Python COMP Superscalar

QDR Quad Data Rate. 14, 19, *Glossary:* Quad Data Rate

RAM Random-Access Memory. 17, 139, 142, 143, *Glossary:* Random-Access Memory

RDMA Remote Direct Memory Access. 38, 144, *Glossary:* Remote Direct Memory Access

ReRAM Resistive Random-Access Memory. 26, 27, *Glossary:* Resistive Random-Access Memory

RoCE Remote Direct Memory Access over Converged Ethernet. 29, *Glossary:* Remote Direct Memory Access over Converged Ethernet

ROCm Radeon Open Compute. 45, 46, 51, *Glossary:* Radeon Open Compute

SDRAM Synchronous Dynamic Random-Access Memory. 12, 14, 17, *Glossary:* Synchronous Dynamic Random-Access Memory

SRAM Static Random-Access Memory. 17, 22, 28, 36, *Glossary:* Static Random-Access Memory

SDR Single Data Rate. 14, 19, *Glossary:* Single Data Rate

SHMEM Symmetric Hierarchical Memory. 102, *Glossary:* Symmetric Hierarchical Memory

SICM Simplified Interface to Complex Memory. 43, 47, 51, 76, 78, 82, 84, 90, 92, 93

SIMD Single-Intruction-Multiple-Data. *Glossary:* Single-Intruction-Multiple-Data

SIMM Single in-line Memory Module. 14, 17, 135, *Glossary:* Single in-line Memory Module

SMP Symmetric Multiprocessing. 102, *Glossary:* Symmetric Multiprocessing

SNIA Storage Networking Industry Association. 28, 33, *Glossary:* Storage Networking Industry Association

SNL Sandia National Laboratory. 43, 44

SSD Solid-State Drive. 26, 29, 32, 141, *Glossary:* Solid-State Drive

STT-MRAM Spin-Transfer Torque Magnetoresistive Random-Access Memory. 26, 27, *Glossary:* Spin-Transfer Torque Magnetoresistive Random-Access Memory

SVM shared virtual memory. 40, *Glossary:* Shared Virtual Memory

TB Terabytes. 26, *Glossary:* Terabytes

TSV through-silicon via. 24

UDJ Universal Data Junction. 4, 72

Bibliography

[1] Testing memory allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc while trying to simulate real-world loads. http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/. Accessed: 2020–10–30.

[2] Supermalloc. https://github.com/kuszmaul/SuperMalloc/tree/master/tests, 2015.

[3] JEDEC Committee 42.3C. Graphics Double Data Rate 6 (GDDR6) SGRAM standard (JESD250B). Technical report, JEDEC, November 2018.

[4] JEDEC Committee 42.3C. DDR4 SDRAM standard (JESD79-4C). Technical report, JEDEC, January 2020.

[5] JEDEC Committee 42.3C. High Bandwidth Memory (HBM) DRAM (JESD235C). Technical report, JEDEC, January 2020.

[6] Hiro Akinaga and Hisashi Shima. Reram technology; challenges and prospects. *IEICE Electronics Express*, 9:795–807, 01 2012.

[7] Javier Álvarez Cid-Fuentes, Pol Álvarez, Ramon Amela, Kuninori Ishii, Rafael K. Morizawa, and Rosa M. Badia. Efficient development of high performance data analytics in Python. *Future Generation Computer Systems*, 2019.

[8] Juan Guillermo Alzate-Vinasco, Umut Arslan, Peng Bai, Justin Brockman, Yu-Jin Chen, Nilanjan Das, Kevin Fischer, Tahir Ghani, Philip Heil, Patrick Hentges, Rawshan Jahan, Aaron Littlejohn, Mohammad Mainuddin, Daniel Ouellette, James Pellegren, Tanmoy Pramanik, Conor Puls, Pedro Quintero, Tofizur Rahman, Meenakshi Sekhar, Bernhard Sell, Mansi Seth, Andrew J. Smith, Angeline K. Smith, Liqiong Wei, Chris Wiegand, Oleg Golonzka, and Fatih

Hamzaoglu. 2 MB array-level demonstration of STT-MRAM process and performance towards L4 cache applications. In *2019 IEEE International Electron Devices Meeting (IEDM)*, pages 2.4.1–2.4.4, December 2019.

[9] Giorgos Apostolidis, Dimitrios Balobas, and Nikos Konofaos. Design and simulation of 6t sram cell architectures in 32nm technology. *Journal of Engineering Science and Technology Review*, 9:145–149, 01 2016.

[10] Lucian Armasu. What we know about DDR5 so far, June 2019.

[11] JEDEC Solid State Technology Association. JEDEC announces support for NVDIMM hybrid memory modules, May 2015.

[12] JEDEC Solid State Technology Association. DDR4 NVDIMM-N design specificaiton. Technical report, JEDEC, March 2018.

[13] JEDEC Solid State Technology Association. JEDEC to hold workshops for DDR5, LPDDR5 & NVDIMM-P standards, September 2019.

[14] JEDEC Solid State Technology Association. Main memory: DDR4 & DDR5 SDRAM. Technical report, JEDEC, 2020.

[15] JEDEC Solid State Technology Association. Main memory: DDR4 & DDR5 SDRAM (press release). Technical report, JEDEC, 2020.

[16] Prateek Asthana and Sangeeta Karyakarte. Performance comparison of 4t, 3t and 3t1d dram cell design on 32nm technology. volume 4, pages 121–133, 07 2014.

[17] Guillaume Aupy, Olivier Beaumont, and Lionel Eyraud-Dubois. What size should your buffers to disks be? In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 660–669, May 2018.

[18] Avalanche Technology. *Endurance, Data Retention and Field Immunity of STT-MRAM*, November 2019. Revision C.

[19] David A. Beckingsale, Marty J. McFadden, Johann P. S. Dahm, Ramesh Pankajakshan, and Richard D. Hornung. Umpire: Application-focused management and coordination of complex hierarchical memory. *IBM Journal of Research and Development*, 64(3/4):00:1–00:10, May 2020.

[20] David Alexander Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, Adam J. Kunen, William Killian, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas R. W. Scogland. RAJA: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, November 2019.

[21] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186, February 2010.

[22] BSC — Workflows and Distributed Computing. K-means with Redis. https://github.com/bsc-wdc/apps/tree/stable/python/examples/kmeans_redis, 2019.

[23] Christopher Cantalupo, Vishwanath Venkatesan, Jeff R. Hammond, Krzysztof Czurylo, and Simon Hammond. User extensible heap manager for heterogeneous memory platforms and mixed memory policies. *Architecture document*, 2015.

[24] Qinglei Cao, George Bosilca, Wei Wu, Dong Zhong, Aurélien Bouteiller, and Jack Dongarra. Flexible data redistribution in a task-based runtime system. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–225, 2020.

[25] Chandler Carruth. Tuning C++: Benchmarks, and compilers, and CPUs! Oh My!, September 2015.

[26] Barcelona Supercomputing Center and Centro National de Supercomputación. MareNostrum 3, 2012.

[27] Kevin Kai-Wei Chang, Lee Donghyuk, Zeshan Chishti, Alaa R. Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM performance by parallelizing refreshes with accesses. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 356–367, February 2014.

[28] Yuan-Hao Chang, Jian-Hong Lin, Jen-Wei Hsieh, and Tei-Wei Kuo. A strategy to emulate NOR flash with NAND flash. *ACM Trans. Storage*, 6(2), July 2010.

[29] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3, 2010.

[30] Jong Chern Lee, Jihwan Kim, Kyung Whan Kim, Young Jun Ku, Dae Suk Kim, Chunseok Jeong, Tae Sik Yun, Hongjung Kim, Ho Sung Cho, Sangmuk Oh, Hyun Sung Lee, Ki Hun Kwon, Dong Beom Lee, Young Jae Choi, Jaejin Lee, Hyeon Gon Kim, Jun Hyun Chun, Jonghoon Oh, and Seok Hee Lee. High bandwidth memory (HBM) with TSV technique. In *2016 International SoC Design Conference (ISOCC)*, pages 181–182, October 2016.

[31] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, June 2016.

[32] Jaeyoung Choi, James Weldon Demmel, Inderjit S. Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Kendall S. Stanley, David Walker, and R. Clint Whaley. ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance. *Computer Physics Communications*, 97(1):1–15, 1996. High-Performance Computing in Science.

[33] Jonathan Corbet. Transparent hugepages in 2.6. 38. *LWN, http://lwn.net/Articles/423584, archived at https://perma.cc/4MRR-49RE*, 2011.

[34] Intel Corporation. oneAPI specification. Online Documentation, 2020. Release 0.85.

[35] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[36] Muthu Dayalan. Resistive random access memory (reram). *International Journal of Research and Engineering*, 6, 06 2019.

[37] Mattias De Wael, Stefan Marr, Bruno Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 47:1–27, May 2015.

[38] Vishal Deep and Tarek Elarabi. Write latency reduction techniques of state-of-the-art phase change memory. In *2016 European Modelling Symposium (EMS)*, pages 213–217, November 2016.

[39] Bin Dong, Suren Byna, Kesheng Wu, Prabhat, Hans Johansen, Jeffrey N. Johnson, and Noel Keen. Data elevator: Low-contention data movement in hierarchical storage system. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 152–161, December 2016.

[40] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Hybrid checkpointing using emerging nonvolatile memories for future exascale systems. *ACM Trans. Archit. Code Optim.*, 8(2), June 2011.

[41] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean–Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Yutaka Ishikawa, Zhong Jin, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias Mueller, Wolfgang Nagel, Hiroshi Nakashima, Michael E. Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad van der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Yelick Kathy. The international exascale software project roadmap. *International Journal of High Performance Computer Applications*, 25(1), January 2011.

[42] Thaleia Dimitra Doudali and Ada Gavrilovska. CoMerge: Toward efficient data placement in shared heterogeneous memory systems. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'17, pages 251–261, New York, NY, USA, October 2017. ACM.

[43] Michael Driscoll, Amir Kamil, Shoaib Kamil, Yili Zheng, and Katherine Yelick. PyGAS: A Partitioned Global Address Space extension for Python. *The Sixth Conference on Partitioned Global Address Space Programming Models (PGAS 2012)*, October 2012. Poster abstract.

[44] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 15:1–15:16, New York, NY, USA, April 2016. ACM.

[45] Tim Dykes, Clément Foyer, Harvey Richardson, Martin Svedin, Artur Podobas, Stefano Markidis, Adrian Tate, Ioan Hadade, Olivier Marsden, and Simon McIntosh-Smith. Mamba: Array-based abstractions for heterogeneous memory systems. [in preparation].

[46] Thomas Edvalson. HIP documentation. Online Documentation, July 2020.

[47] H. Carter Edwards and Christian R. Trott. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*, pages 18–24, 2013.

[48] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[49] Sean Eilert, Mark Leinwander, and Giuseppe Crisenza. Phase change memory: A new memory enables new memory usage models. In *2009 IEEE International Memory Workshop*, pages 1–2, May 2009.

[50] Jason Evans. A scalable concurrent malloc(3) implementation for FreeBSD. *FreeBSD presentations and papers*, April 2006.

[51] Everspin Technologies, Inc. 256 Mbit *ST-DDR3 Spin-transfer Torque MRAM*, October 2018. Revision 1.3.

[52] Everspin Technologies, Inc. 1 GB *ST-DDR4 Spin-transfer Torque MRAM*, February 2020. Revision 1.1.

[53] Pradeep Fernando, Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. Phoenix: Memory speed HPC I/O with NVM. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 121–131, December 2016.

[54] UEFI Forum. Advanced configuration and power interface specification. Technical report, UEFI Forum, May 2017. Version 6.2.

[55] Clément Foyer, Adrian Tate, and Simon McIntosh-Smith. ASPEN: An efficient algorithm for data redistribution between producer and consumer grids. In Gabriele Mencagli, Dora B. Heras, Valeria Cardellini, Emiliano Casalicchio, Emmanuel Jeannot, Felix Wolf, Antonio Salis, Claudio Schifanella, Ravi Reddy Manumachu, Laura Ricci, Marco Beccuti, Laura Antonelli, José Daniel Garcia Sanchez, and Stephen L. Scott, editors, *Euro-Par 2018: Parallel Processing Workshops*, pages 171–182, Cham, 2019. Springer International Publishing.

[56] Fujitsu Limited. *A64FX®, Microarchitecture manual*, April 2020. English version, Revision 1.1.

[57] Bill Gervasi and Jonathan Hinkle. Overcoming system memory challenges with persistent memory and NVDIMM-P. JEDEC Server Forum 2017, June 2017.

[58] Brice Goglin. Towards the structural modeling of the topology of next-generation heterogeneous cluster nodes with hwloc. Research report, Inria, November 2016.

[59] SNIA NVM Programming Technical Working Group. NVM programming model (NPM) version 1.2. SNIA technical position, The Storage Networking Industry Association (SNIA), June 2017.

[60] Minyi Guo and Ikuo Nakata. A framework for efficient data redistribution on distributed memory multicomputers. *The Journal of Supercomputing*, 20(3):243–265, November 2001.

[61] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR'17, pages 16:1–16:9, New York, NY, USA, May 2017. ACM.

[62] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3D XPoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[63] Stephen Herbein, Dong H. Ahn, Don Lipari, Thomas R. W. Scogland, Marc Stearman, Mark Grondona, Jim Garlick, Becky Springmeyer, and Michela Taufer. Scalable I/O-aware job scheduling for burst buffer enabled HPC clusters. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'16, pages 69–80, New York, NY, USA, June 2016. ACM.

[64] Rich Hornung, Holger Jones, Jeff Keasler, Rob Neely, Olga Pearce, Si Hammond, Christian Trott, Paul Lin, Courtenay Vaughan, Jeanine Cook, Rob Hoekstra, Ben Bergen, Josh Payne, and Geoff Womeldorff. ASC tri-lab co-design level 2 milestone report 2015. Technical report, U.S. Department of Energy, Office of Scientific and Technical Information, September 2015.

[65] Ching-Hsien Hsu, Sheng-Wen Bai, Yeh-Ching Chung, and Chu-Sing Yang. A generalized basic-cycle calculation method for efficient array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1201–1216, December 2000.

[66] Ching-Hsien Hsu, Yeh-Ching Chung, Don-Lin Yang, and Chyi-Ren Dow. A generalized processor mapping technique for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):743–757, July 2001.

[67] Lei Huang, Haoqiang Jin, Liqi Yi, and Barbara Chapman. Enabling locality-aware computations in OpenMP. *Scientific Programming*, 18:169–181, January 2010.

[68] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module, 2019.

[69] JEDEC Committee JC-45.6. DDR4 NVDIMM-P BUSS PROTOCOL, November 2020.

[70] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*

2nd Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2016.

[71] Hongshin Jun, Jinhee Cho, Kangseol Lee, Ho-Young Son, Kwiwook Kim, Hanho Jin, and Keith Kim. HBM (High Bandwidth Memory) DRAM technology and architecture. In *2017 IEEE International Memory Workshop (IMW)*, pages 1–4, May 2017.

[72] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS design for heterogeneous memory management in datacenter. *SIGARCH Comput. Archit. News*, 45(2):521–534, June 2017.

[73] Dounia Khaldi and Barbara Chapman. Towards automatic HBM allocation using LLVM: A case study with knights landing. In *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, pages 12–20, November 2016.

[74] The Khronos Group, Inc. *The OpenCL™ Specification*, April 2020.

[75] The Khronos Group, Inc. *SYCL™ Specification, SYCL™ integrates OpenCL™ devices with modern C++*, April 2020. Version 1.2.1, Document revision 7.

[76] SangBum Kim, Yuan Zhang, James P. McVittie, Hemanth Jagannathan, Yoshio Nishi, and H.-S. Philip Wong. Integrating phase-change memory cell with Ge nanowire diode for crosspoint memory—experimental demonstration and analysis. *IEEE Transactions on Electron Devices*, 55(9):2307–2313, 2008.

[77] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. *SIGARCH Comput. Archit. News*, 40(3):368–379, June 2012.

[78] Peter M. Kogge. The architecture of pipelined computers. In *McGraw-Hill advanced computer science series*, pages 321–328, 1981.

[79] Nasser Kurd, Muntaquim Chowdhury, Edward Burton, Thomas P. Thomas, Christopher Mozak, Brent Boswell, Praveen Mosalikanti, Mark Neidengard, Anant Deval, Ashish Khanna, Nasirul Chowdhury, Ravi Rajwar, Timothy M. Wilson, and Rajesh Kumar. Haswell: A family of IA 22 nm processors. *IEEE Journal of Solid-State Circuits*, 50(1):49–58, January 2015.

[80] Michael Lang, Latchesar Ionkov, and Sean Williams. Simplified interface to complex memory hierarchies 1.x, version 00, February 2017.

[81] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. volume 34, pages 176–185. Press, January 1998.

[82] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. *SIGARCH Comput. Archit. News*, 37(3):2–13, June 2009.

[83] Daan Leijen. Mimalloc-bench. https://github.com/daanx/mimalloc-bench, 2020.

[84] Daan Leijen, Ben Zorn, and Leonardo de Moura. Mimalloc: Free list sharding in action. Technical Report MSR-TR-2019-18, Microsoft, June 2019.

[85] Edgar A. León, Brice Goglin, and Andres Rubio Proaño. M&MMs: Navigating complex memory spaces with hwloc. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'19, pages 149–155, New York, NY, USA, 2019. ACM.

[86] Chuck Lever and David Boreham. Malloc() performance in a multithreaded linux environment. In *USENIX Annual Technical Conference, FREENIX Track*, 2000.

[87] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern DRAM devices: Implications for retention time profiling mechanisms. *SIGARCH Comput. Archit. News*, 41(3):60–71, June 2013.

[88] Jamie Liu, Ben Jaiyen, Richard Veras, and Onur Mutlu. RAIDR: Retention-aware intelligent DRAM refresh. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, June 2012.

[89] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M. Badia. ServiceSs: An interoperable programming framework for the cloud. *Journal of grid computing*, 12(1):67–91, 2014.

[90] David B. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25–42, February 1993.

[91] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[92] Macronix International Co., Ltd. *Program/Erase Cycling Endurance and Data Retention of Macronix SLC NAND Flash Memories*, October 2014. Technical Note, Rev. 1, Oct. 15, 2014.

[93] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 126–136, February 2015.

[94] Micron Technology, Inc. *TN-12-30: NOR Flash Cycling Endurance and Data Retention*, November 2017. Technical Note, Rev. C 11/17 EN.

[95] Micron Technology, Inc. *Hybrid Memory Cube — HMC Gen2, MT43A4G40200 — 2 GB 4H DRAM stack*, February 2018. Rev H 02/18 EN.

[96] Micron Technology, Inc. *8 GB: ×4, ×8, ×16 DDR4 SDRAM*, April 2020. Rev R 04/2020 EN.

[97] Mathieu Mirmont. Python extension: SharedArray. https://pypi.org/project/SharedArray/, 2019.

[98] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015.

[99] Andy Morris. Breaking boundaries with supercomputer storage, November 2019.

[100] Mouser Electronics, Inc. Hard drive cost per gigabyte. URL: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/, July 2017.

[101] Mouser Electronics, Inc. NOR Flash prices. URL: https://www.mouser.co.uk/Semiconductors/Memory-ICs/NOR-Flash/_/N-488w1, 2020.

[102] Prashant J. Nair, Chiachen Chou, Bipin Rajendran, and Moinuddin K. Qureshi. Reducing read latency of phase change memory via early read and turbo read. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 309–319, February 2015.

[103] Dan Negrut, Radu Serban, Ang Li, and Andrew Seidl. Unified memory in CUDA 6: a brief overview of related data access and transfer issues. *Tech. Rep. TR-2014-09, University of Wisconsin-Madison, 2014*, June 2014.

[104] NVIDIA. *CUDA C++ programming guide*, July 2020. Design Guide, PG-02829-001_v11.0.

[105] Paramjit Oberoi and Gurindar Sohi. Out-of-order instruction fetch using multiple sequencers. In *In Proceedings of the Intl. Conference on Parallel Processing*, pages 14–26, 2002.

[106] Lena Oden and Pavan Balaji. Hexe: A toolkit for heterogeneous memory management. *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 656–663, 2017.

[107] Stephen L. Olivier, Allan K. Porterfield, Kyle B. Wheeler, Michael Spiegel, and Jan F. Prins. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications*, 26(2):110–124, 2012.

[108] OpenMP Architecture Review Board. OpenMP application programming interface specification.

[109] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, November 2018.

[110] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Sixth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2019.

[111] Antonio J. Peña and Pavan Balaji. Toward the efficient use of multiple explicitly managed memory subsystems. In *2014 IEEE International Conference On Cluster Computing (CLUSTER)*, CLUSTER'14, pages 123–131, September 2014.

[112] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. System evaluation of the Intel Optane byte-addressable NVM. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'19, pages 304–315, New York, NY, USA, October 2019. ACM.

[113] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Pietro Cicotti, Erwin Laure, and Stefano Markidis. RTHMS: A tool for data placement on hybrid memory system. *SIGPLAN Not.*, 52(9):82–91, June 2017.

[114] Swann Perarnau, Brice Videau, Nicolas Denoyelle, Florence Monna, Kamil Iskra, and Pete Beckman. Explicit data layout management for autotuning exploration on complex memory topologies. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, MCHPC'19, pages 63–68, New York, NY, USA, 2019. Association for Computing Machinery.

[115] Antoine P. Petitet and Jack J. Dongarra. Algorithmic redistribution methods for block-cyclic decompositions. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1201–1216, December 1999.

[116] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. PARAVER: A tool to visualize and analyze parallel code. *WoTUG-18*, 44, March 1995.

[117] Loïc Prylli and Bernard Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *J. Parallel Distrib. Comput.*, 45(1):63–72, August 1997.

[118] Shankar Ramasulamy and Prithviraj Banerjee. Automatic generation of efficient array redistribution routines for distributed memory multicomputers. In *The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 342–349, February 1995.

[119] Shankar Ramaswamy, Barbara Simons, and Prithviraj Banerjee. Optimizations for efficient array redistribution on distributed memory multicomputers. *Journal of Parallel and Distributed Computing*, 38(2):217–228, November 1996.

[120] Adrien Roussel, Patrick Carribault, and Julien Jaeger. Preliminary Experience with OpenMP Memory Management Implementation. In *IWOMP 2020*, Austin, United States, September 2020.

[121] Yole Development SA. Emerging NVM (Non-Volatile Memory) technologies & markets 2015, 2015.

[122] Toshiyuki Shimizu. Post-K supercomputer with Fujitsu's original CPU, A64FX powered by Arm ISA, November 2018.

[123] Storage Networking Industry Association (SNIA). Persistent memory summit, January 2018.

[124] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat R. Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, March 2016.

[125] Enric Tejedor, Yolanda Becerra, Guillem Alomar, Anna Queralt, Rosa M. Badia, Jordi Torres, Toni Cortes, and Jesús Labarta. PyCOMPSs: Parallel computational workflows in Python. *The International Journal of High Performance Computing Applications*, 31(1):66–82, 2017.

[126] François Tessier, Paul Gressier, and Venkatram Vishwanath. Optimizing data aggregation by leveraging the deep memory hierarchy on large-scale systems. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS'18, pages 229–239, New York, NY, USA, June 2018. ACM.

[127] François Tessier, Maxime Martinasso, Matteo Chesi, Mark Klein, and Miguel Gila. Dynamic Provisioning of Storage Resources: A Case Study with Burst Buffers. In *IPDPSW 2020 — IEEE International Parallel and Distributed Processing Symposium Workshops*, New Orleans, United States, May 2020.

[128] Rajeev Thakur, Alok Choudhary, and Geoffrey Fox. *Runtime array redistribution in HPF programs*, pages 309–316. IEEE, December 1994.

[129] Rajeev Thakur, Alok Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594, June 1996.

[130] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L. Chamberlain, Romain Cledat, H. Carter Edwards, Hal Finkel, Karl Fuerlinger, Frank Hannig, Emmanuel Jeannot, Amir Kamil, Jeff Keasler, Paul H. J. Kelly, Vitus Leung, Hatem Ltaief, Naoya Maruyama, Chris J. Newburn, and Miquel Pericás. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020, 2017.

[131] Stéfan J. van der Walt, S. Chris Colbert, and Gaël Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[132] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. An early evaluation of Intel's Optane DC Persistent Memory Module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'19, pages 76:1–76:19, New York, NY, USA, November 2019. ACM.

[133] Maurice V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, EC-14(2):270–271, April 1965.

[134] Sean Williams. Simplified interface to complex memory. In *Exascale Computing Project Kick-off meeting*, 2017.

[135] Sean Williams, Latchesar Ionkov, and Michael Lang. NUMA distance for heterogeneous memory. In *Proceedings of the Workshop on Memory Centric Programming for HPC*, MCHPC'17, pages 30–34, New York, NY, USA, November 2017. Association for Computing Machinery.

[136] Sean Williams, Latchesar Ionkov, Michael Lang, and Jason Lee. Heterogeneous memory and arena-based heap allocation. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, MCHPC'18, pages 67–71, New York, NY, USA, 2018. Association for Computing Machinery.

[137] H.-S. Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P. Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Ken-

neth E. Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, December 2010.

[138] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[139] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring performance characteristics of the Optane 3D XPoint storage technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 5(1), February 2020.